EFFICIENT EUCLIDEAN DISTANCE CALCULATIONS AND DISTANCE SIMILARITY SEARCHES: AN EXAMINATION OF HETEROGENEOUS CPU, GPU, AND TENSOR CORE ARCHITECTURES

By Benoît Gallet

A Dissertation
Submitted in Partial Fulfillment
of the Requirements for the Degree of
Doctor of Philosophy
in Informatics and Computing

Northern Arizona University
May 2023

Approved:
Michael Gowanlock, Ph.D., Chair
Frédéric Loulergue, Ph.D.
Toby Hocking, Ph.D.
Dieter Otte, Ph.D.
Yiqi Luo, Ph.D.

ABSTRACT

EFFICIENT EUCLIDEAN DISTANCE CALCULATIONS AND DISTANCE SIMILARITY SEARCHES: AN EXAMINATION OF HETEROGENEOUS CPU, GPU, AND TENSOR CORE ARCHITECTURES

BENOÎT GALLET

The Euclidean distance is a measure frequently used in numerous applications, including data-analysis algorithms, to determine the similarity between objects, as a function of the distance between them. Given a set of objects, performing a distance similarity search consists of finding objects that are considered similar, i.e., finding objects within a threshold search distance of a given object, where the distance measure often employs the Euclidean distance formula. Distance similarity searches can be used as a building block for other algorithms, including the distance similarity join, k-Nearest Neighbors (k-NN), k-Means, or the Density-Based Spatial Clustering of Applications with Noise (DBSCAN) algorithms. As such, optimizing the computation of Euclidean distances will improve the performance of distance similarity searches, and improving distance similarity searches will improve the performance of numerous other algorithms. Consequently, optimizing both Euclidean distance calculations and distance similarity searches is critical to improve the performance of many data-analysis algorithms, including the ones mentioned above, in addition to applications in other domains such as fields that require modeling and simulation.

The literature is rich with methods to improve the performance of Euclidean distance calculations and distance similarity searches, particularly using Central Processing Units (CPUs). While multicore CPUs can offer great parallel performance, they are outclassed by the higher computational throughput of Graphics Processing Units (GPUs). Using GPUs for these problems is relatively recent and there are, consequently, significantly fewer proposed work that use the GPU instead of the CPU. However, the design space for GPU algorithms is large, thus some algorithm designs have been neglected, including those that carefully exploit GPU resources. Furthermore, while both CPUs and GPUs have been extensively studied on their own, very little work has been conducted where both architectures are leveraged concurrently.

Tensor Cores (TCs) are a recent addition to certain GPU architectures. As an Application-Specific Integrated Circuit (ASIC), TCs are designed to compute Matrix Multiply-Accumulate (MMA) operations, at a higher throughput than other general-purpose cores. In the literature, TCs are primarily used for machine learning and other related fields involving linear algebra, yielding great performance improvements. Despite their specificity, TCs can be leveraged for any algorithm where the computation can be expressed using MMA operations. Nevertheless, leveraging TCs for general-purpose scientific algorithms remains an open problem.

We propose in this dissertation to optimize the performance of Euclidean distance calculations and more generally distance similarity searches, by examining: (*i*) GPU resource utilization; (*ii*) the joint use of both CPUs and GPUs for computation; (*iii*) the use of TCs to compute Euclidean distances; (*iv*) the joint use of general-purpose GPU cores and TCs to compute Euclidean distances.

ACKNOWLEDGEMENTS

First and foremost, I would like to show my gratitude to my supervisor, Michael Gowanlock. He helped and supported me even before starting this Ph.D., never stopped since, and always provided me with incredibly valuable advice. I am very grateful and glad I got to be his student and to learn from him. Thank you for accepting to be my supervisor from the beginning and until the end!

I would then like to thank the members of my committee: Frédéric Loulergue, Toby Hocking, Dieter Otte, and Yiqi Luo for accepting this task in the first place, for reading this manuscript, and for their very useful recommendations on how to make it better.

Many thanks to all the people I have befriended while in Flagstaff, including my previous and current housemates. There are too many of you to name everyone individually, but know that your support and the time spent with you all these years were extremely appreciated, and sometimes even needed!

Finally, I would like to thank Ciarán, whom I only met this year but who is already important to me, and who greatly supported me in this last stretch. Thank you for being there, and I hope that you will be by my side for a long time.

REMERCIEMENTS

J'aimerais tout d'abord remercier Frédéric Loulergue sans qui toute cette aventure n'aurait tout simplement pas été possible. D'Orléans à Flagstaff, c'est grâce à lui que j'en suis ici aujourd'hui. Merci également pour les précieux conseils au fil des années et au moment de la relecture de ce manuscrit.

J'aimerais ensuite remercier ma famille, et plus particulièrement ma mère ainsi que mon beau-père pour leur soutien depuis l'autre côté du globe. Maman, je me souviendrai toujours de ce samedi matin ou je vous ai annoncé que j'étais accepté dans ce programme, et que j'allais partir plusieurs années à des milliers de kilomètres. Tes mots ont été *"ton père aurait été fier de toi"*. Je l'espère, et j'espère que vous l'êtes aussi. Merci également à mes frères, ma sœur ainsi que ma grand-mère.

À Emmanuel et Martin, merci pour ces années passées à vos côtés, les projets de l'université, votre soutien, et les innombrables heures à jouer ensemble (peut-être même trop !).

Et pour conclure ces remerciements: Jolan, présent depuis le début. Le lycée, l'université, Flagstaff. Tu es à mes côtés depuis de nombreuses années à m'aider, me soutenir, à vivre des aventures que je n'oublierai jamais. En particulier la première année à Flagstaff ! Anthony et Ana, également présents depuis des années déjà. Chaque moment avec vous est fantastique, inattendu, et toujours mémorable. Merci pour votre soutien, vos encouragements, ainsi que les aventures vécues et à venir.

Je ne pourrais pas rêver d'une meilleur famille et de meilleurs amis que ce que j'ai. Je vous aime.

Table of Contents

List of Tables

List of Figures

LIST OF ABBREVIATIONS AND DEFINITIONS

Abbreviations

AMX Advanced Matrix Extensions

ASIC Application-Specific Integrated Circuit

AVX Advanced Vector Extensions

CPU Central Processing Unit

CUDA Compute Unified Device Architecture

DBSCAN Density-Based Spatial Clustering of Applications with Noise

FP16 Half-precision floating-point format using 16 bits

FP32 Single-precision floating-point format using 32 bits

FP64 Double-precision floating-point format using 64 bits

GPGPU General-purpose computing on Graphics Processing Units

GPU Graphics Processing Unit

ILP Instruction-Level Parallelism

***k*-NN** *k*-Nearest Neighbors

MMA Matrix Multiply-Accumulate

RT Ray-tracing

SIMD Sinhle Instruction Multiple Data

SIMT Single Instruction Multiple Threads

TC Tensor Core

WMMA Warp Matrix Multiply-Accumulate

Definitions

Candidate point Point to which we compute the distance to a query point to determine if it is within a search threshold.

Catastrophic cancellation Extreme loss of precision when subtracting two nearby floating point numbers or when adding two nearby floating point numbers of opposite signs. This issue is more likely to happen when using lower precision data types, such as data types stored using 16 bits.

Clustering Application that groups objects together.

CUDA API designed by Nvidia to give programmatic access to their GPUs.

CUDA Cores Smallest compute unit of Nvidia GPUs.

Distance similarity join Algorithm finding all pairs of points that are within a distance ϵ from each other.

DBSCAN Clustering algorithm identifying an unspecified amount of clusters that have a minimum density, and discarding points considered as noise.

Global memory Dedicated off-chip GPU memory, with the largest capacity but also highest bandwidth and latency than other GPU memories.

Heterogeneous computing Involves concurrently using different architectures (e.g., CPU and GPU) for computing a single problem. We consider here that heterogeneous computing requires the CPU to act as more than just a host for the GPU.

Host Role endorsed by the CPU when also using a GPU. Its role consists of allocating memory on the GPU, initiating the memory transfers between the CPU and GPU, start the kernels, etc. A host is mandatory when using the GPU.

ILP Instruction-Level Parallelism, ability for a processor's core to compute several independent instructions at once, such as an addition and a multiplication on different variables.

Indexing Method used to store objects usually based on their spatial coordinates. Indexing structures are then used to efficiently retrieve specific points (typically candidate points), and can greatly reduce the number of points to search compared to brute-force searches.

Kernel Function instantiated by the host and executed on the GPU.

***k*-NN** Algorithm finding the k nearest neighbors of a query point, according to a distance measure.

Main memory RAM available to the CPU.

Query point The point being searched within a given algorithm.

Range query Ensemble of the distance calculations between a query point and its candidate points.

Ray Tracing Cores Type of specialized GPU core designed to improve the performance of ray tracing. These units are designed to efficiently search Bounding Volume Hierarchy trees and compute intersection operations.

Search-and-refine Execution model where an index is searched for each query point to prune their search space and reduce the number of candidate points. The candidate points are then refined to compute the actual result.

SIMD Single Instruction Multiple Data, a parallel execution model where multiple threads execute the same instruction, but on different data elements. SIMD is also possible on a single thread when using vectorized instructions.

SIMT Single Instruction Multiple Threads, a parallel execution model used by GPUs, where the same instruction is executed concurrently by the threads of a warp.

Tensor Cores Type of specialized GPU core designed for Matrix Multiply-Accumulate (MMA) operations.

Thread divergence Occurs when GPU threads of a warp do not follow the same path in the instruction control flow, such as in conditional statements. Branches are executed sequentially, reducing the overall parallelism and thus performance.

Warp Group of 32 GPU threads that execute the same instruction, and which is defined in hardware.

WMMA Warp Matrix Multiply-Accumulate, library provided by Nvidia giving a programmatic access to the Tensor Cores. This also describes a computing model, where a GPU warp calculates a matrix-multiply accumulate operation, where two matrices are multiplied and a third matrix is added to the result.

PREFACE

This dissertation contains four peer-reviewed publications. We summarize these publications as follows:

- Chapter 3: Benoit Gallet and Michael Gowanlock, Load Imbalance Mitigation Optimizations for GPU-Accelerated Similarity Joins, *IEEE High-Performance Big Data, Deep Learning, and Cloud Computing Workshop (HPBDC), Proceedings of the 2019 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 396–405, 2019, doi: 10.1109/IPDPSW.2019.00078. *Note: This paper won the best paper award at the High-Performance Big Data and Cloud Computing Workshop.*

- Chapter 4:

  - Benoit Gallet and Michael Gowanlock, HEGJoin: Heterogeneous CPU-GPU Epsilon Grids for Accelerated Distance Similarity Join, *Proceedings of the* 25^{th} *International Conference on Database Systems for Advanced Applications (DASFAA)*, pages 372–388, 2020, doi: 10.1007/978-3-030-59419-0 23.

  - Benoit Gallet and Michael Gowanlock, Heterogeneous CPU-GPU Epsilon Grid Joins: Static and Dynamic Work Partitioning Strategies, *Data Science and Engineering*, volume 6, pages 39–62, 2020, doi: 10.1007/s41019-020-00145-x.

  - Note that we include in this dissertation the extended journal version of the conference paper.

- Chapter 5: Benoit Gallet and Michael Gowanlock, Leveraging GPU Tensor Cores for Double Precision Euclidean Distance Calculations, *Proceedings of the* 29^{th} *IEEE International Conference on High Performance Computing, Data, and Analytics (HiPC)*, pages 135–144, 2022, doi: 10.1109/HiPC56025.2022.00029.

Chapter 6 is formatted as a conference paper, but has not been accepted for publication yet.

Chapter 1

Introduction

In recent years, there has been a need to process or analyze large volumes of data using efficient algorithms. Similarity searches are one such algorithm that is often employed to process large datasets, and are widely used in various fields such as image [30] and document similarity [12], or data clustering [19, 20, 32]. More generally, similarity search algorithms are typically used to find objects in a dataset that are similar to a given query object, and according to a given metric. The similarity between two objects corresponds to their computed distance, and two objects are considered similar if their distance is within a certain threshold, which is defined by the user and may vary depending on the use case. Using traditional processing techniques, which are typically sequential by default, severely limit the performance of similarity search algorithms.

Parallel computing consists of processing several tasks or a task on different data at the same time and is an efficient solution to improve similarity search algorithms performance. By breaking down their execution into smaller tasks and executing them concurrently on multiple processing units, whether using Central Processing Units (CPUs) or Graphics Processing Units (GPUs), we can significantly improve the performance of similarity search algorithms, and can process increasingly larger datasets as well. Despite being designed for graphic applications, GPUs recently emerged as a more efficient solution for parallel processing than CPUs, in part thanks to their substantially higher core count and much higher memory bandwidth. More generally, parallel computing is essential to improve the performance of many algorithms, or to allow the processing of larger datasets while yielding a practical execution time.

Similarity search algorithms benefit greatly from parallel processing techniques, as their overall computation often consists of many smaller operations that can be done in parallel, and that therefore scale well with the increased core counts of GPUs compared to CPUs. Additionally, as these algorithms typically process large volumes of data, the higher memory bandwidth of GPUs compared to the system's main memory accessed by the CPU, also helps improve the performance of similarity search algorithms. Consequently, we observe in the literature that GPUs have been gradually more used to implement similarity search algorithms, and are often more efficient than previously proposed algorithms using CPUs.

Many data analysis algorithms, including similarity search algorithms, use the Euclidean distance formula to measure the similarity between pairs of objects [12, 19, 20, 21, 24, 32, 46, 57, 82, 91]. The Euclidean distance measures the straight-line distance between two objects according to their spatial coordinates, and is not constrainted by the number of data dimensions. When many objects are processed, computing Euclidean distances can be a major performance bottleneck, which needs to be addressed. Given a dataset V, computing the similarity between all objects has a time complexity of $O(|V|^2)$ [31]. In the literature, many published works propose to reduce this complexity by reducing the number of distance calculations computed, particularly by using an indexing data structure. Such data structures index data, typically based on their spatial coordinates, which allows for fast data retrieval. More particularly, and depending on the search threshold, indexes ensure that some points can not be within that threshold of a query point, thus typically reducing the overall number of distance calculations to compute, and therefore reducing the overall time complexity of the algorithm. We identify two types of indexing structures, based on trees [13, 14, 16, 17, 33, 48, 87], and grids [21, 22, 45, 46, 57]. Due to the curse of dimensionality [15, 50, 51], many of the indexing structures are efficient only in low or high dimensionality. Furthermore, because these indexing structures are designed for either the CPU or the GPU, and because of the differences between CPU and GPU architectures, algorithms using these structures on the CPU will not perform well on the GPU, and vice versa.

In the literature, the majority of proposed distance similarity search algorithms, or algorithms employing Euclidean distance calculations, examine the CPU as the target platform. With the recent introduction of General Purpose Computing on Graphics Processing Units (GPGPU), GPUs have been increasingly used for a wide range of applications thanks to their significantly greater throughput than CPUs: GPUs typically have several orders of magnitude more cores than CPUs, and their onboard global memory bandwidth is also several times greater than the main memory (RAM). However, where CPU cores can compute independently, the Single Instruction Multiple Thread (SIMT) architecture of GPUs imposes groups of 32 threads to execute the same instruction at a time. Consequently, some algorithms may not perform well when executed on the GPU, particularly algorithms with data dependencies or those with a task-parallel division of work. In the case of Euclidean distances, computing the distance between two objects is independent of computing the distance between two other objects. Thus, computing Euclidean distances is a highly parallel calculation scaling well with the number of threads/cores used. As such, GPUs with thousands of cores are excellent architectures for improving the performance of Euclidean distance calculations, and other algorithms that employ them, including distance similarity searches.

While CPU architectures have not extensively changed in recent years, each consecutive generation of GPU is significantly different and much faster than the previous generation. In particular, the Volta generation of Nvidia GPUs [71] introduced Tensor Cores (TCs). TCs are an Application-Specific Integrated Circuit (ASIC) designed to exclusively compute Matrix Multiply-Accumulate operations, at a much higher throughput than general-purpose CPU or GPU CUDA cores. Given four matrices A, B, C and D, TCs compute $D = A \times B + C$, with a significantly greater throughput than general-purpose cores.

Note that the GPU-related research was conducted using Nvidia GPUs and the Nvidia CUDA API. As such, we use the corresponding terminology throughout this dissertation. Nevertheless, the GPU concepts we discuss in this dissertation are relevant to other modern GPUs as well, and only the terminology may differ.

1.1 Motivations

The performance of data-analysis algorithms is directly dependent on the input data characteristics: the number of objects, their distribution in the data-space, data dimensionality, among other factors. To process increasingly larger datasets in a reasonable amount of time, improving the algorithms is necessary, and the improvements should be able to exploit newer hardware features and, overall, all computational capabilities of a computer platform. The literature regarding Euclidean distance calculation and distance similarity searches has several shortcomings that motivated the work conducted in this dissertation, which we detail below.

- Many research works focus on using either the CPU or the GPU. While focusing on a single architecture makes optimizing algorithms relatively easier, this leaves one or the other architecture underutilized: CPU algorithms do not leverage GPUs by default; GPU algorithms sporadically need to use the CPU as a host for the GPU, essentially performing tasks such as transferring data from main memory (RAM) to the GPU's global memory, and vice versa. As compute clusters and consumer-grade computers are increasingly equipped with GPUs [89], algorithms designed for either the CPU or the GPU would not be able to efficiently leverage all the computational power of such platforms.

- Parallel CPU and GPU algorithms are notoriously challenging to design and optimize. Furthermore, because of all the architectural differences between CPUs and GPUs, efficient optimizations working well on the CPU may not work as well on the GPU, and vice versa. In particular, algorithms with irregular workloads can have a negative impact on the performance, particularly on the GPU because of the SIMT execution model. Distance similarity searches, when using an indexing structure, are an algorithm with such irregular workloads: all the objects may not compare to the same number of objects, where each object may have a different workload. On the SIMT architecture of the GPU, it might yield lower resource usage, as threads with lower workloads might wait on threads with a higher workload to finish their assigned work, effectively reducing the overall parallelism of the algorithm, and thus performance.

- GPU architectures have greatly evolved, and even recent algorithms may not leverage the full capabilities of recent GPUs, or recent features may not be an obvious choice when designing an algorithm for the architecture. TCs fall into this category: fields related to machine learning or more generally to linear algebra extensively use TCs, with groundbreaking efficiency. On the other hand, very few general scientific applications have leveraged TCs to improve their performance, leaving a part of the compute resource unused.

- A CPU algorithm and a GPU algorithm that have the same purpose, e.g., computing kNN searches, may have different workloads depending on how the algorithms are implemented, usually to fit the targeted architecture. When targeting both the CPU and GPU, this workload disparity is likely to make it more difficult to optimally assign the work to the processors: an input object may have a different workload depending on if the CPU processes it, or if the GPU does.

1.2 Challenges

Efficiently computing Euclidean distances and distance similarity searches on different architectures can be challenging. In particular, we identify the following challenges that must be resolved:

- The architecture of the CPU and GPU are very different, and an algorithm designed for the CPU is likely to perform poorly on the GPU, and vice versa. Leveraging both the CPU and GPU at the same time thus requires careful considerations to achieve the best performance.

- GPU architectures are greatly evolving, and are becoming heterogeneous: along with the general-purpose cores, recent GPU architectures are also equipped with cores for a specific purpose, such as TCs. Because these two types of cores serve a different purpose, leveraging them both concurrently also requires particular considerations, similarly to leveraging the CPU and GPU at the same time.

- Algorithms with irregular workloads such as distance similarity searches are more challenging to optimize than algorithms with regular workloads. Irregular workloads between GPU threads lower the overall parallelism, and thus performance, of an algorithm.

1.3 Contributions

This dissertation makes the following contributions, aiming at solving the challenges described above:

- Using an existing state-of-the-art GPU distance similarity search algorithm as a baseline, we identify a major bottleneck regarding the irregular workloads that we solve by reordering the way objects are processed and how objects are assigned to threads, effectively increasing resource usage and performance.

- We combine the previously optimized algorithm and a state-of-the-art parallel CPU distance similarity search algorithm to propose a heterogeneous CPU-GPU algorithm, where both architectures are concurrently used and perform similar tasks (i.e., both the CPU and GPU search an indexing data structure, compute Euclidean distances, etc.). This CPU-GPU algorithm was able to achieve similar or better performance than CPU-only and GPU-only algorithms in most experiments.

- We propose several Euclidean distance calculation algorithms leveraging TCs, which can then be used in other algorithms such as distance similarity searches. By using TCs, we achieve better performance than when using general-purpose CPU or GPU cores.
- While still a work in progress, we investigate leveraging both the general-purpose GPU cores and TCs. By using both types of cores, we expect to make better use of the available resources and to improve the performance compared to algorithms using only one or the other type of core.

Distance similarity searches and, more generally Euclidean distance calculations, are omnipresent in many data analysis algorithms, where they are often responsible for producing the greatest. While most of the literature focuses on optimizing an algorithm for a specific architecture, this may leave other processors of a different architecture unused in a given computer platform. Leveraging all available architectures, including CPUs, general-purpose GPU cores and specific-purpose GPU TCs is essential for improving the overall performance of distance-centric algorithms, and to make better use of available computational resources.

1.4 Outline

This dissertation is outlined as follows: we introduce in Chapter 2 general background necessary for the rest of the dissertation. We then present in Chapter 3 through Chapter 6 the different contributions we make. We finally conclude the dissertation in Chapter 7, where we propose several future research directions as well.

Chapter 2

Background and Related Work

We give in this chapter important background material relevant to the rest of the dissertation. Note that there is substantial overlap with the background sections of subsequent chapters. Furthermore, because Chapters 3-6 are self-contained published or ready to submit papers, some notations used in this section may be inconsistent throughout the dissertation, for which we apologize to the reader.

2.1 Euclidean Distance

Let V be a dataset in d dimensions, where for a point $p \in V$, p_i is the i^{th} point in V where $i = 1, \dots, |V|$, and $p_i(j)$ is the j^{th} coordinate of p where $j = 1, \dots, d$. Given two points $a, b \in V$, the Euclidean distance formula calculates the distance between the points a and b as follows:

$$dist(a,b) = \sqrt{\sum_{j=1}^{d}(a(j) - b(j))^2}. \tag{2.1}$$

The Euclidean distance function is a fundamental operation in many data analysis algorithms, and is heavily used for distance similarity searches [19, 20, 21, 32, 42, 46, 57, 82]. As such, the algorithms that we optimize and that we propose in this dissertation use this function (Equation 2.1) as well. Additionally, we use in Chapters 5 and 6 an expanded form of the Euclidean distance formula:

$$dist(a,b) = \sqrt{\sum_{j=1}^{d} a(j)^2 - 2a(j)b(j) + b(j)^2} \tag{2.2}$$

2.2 Distance Similarity Searches

Let V and V' be datasets in d dimensions. A distance similarity search consists of, for a query point $q \in V$, to find all the points $c_i \in V'$, where $i = 1, \dots, |V'|$ and $dist(q, c_i) \leq \epsilon$, most commonly using the Euclidean distance function (Equation 2.1) defined above. This computation can also be referred to as a range query. Finding all pairs of points in V and V' that are within ϵ to each other corresponds to computing a distance similarity join ($V \ltimes_\epsilon V'$), and where those pairs are stored in a result set R. Note the special case where V and V' are the same datasets, and thus where $V \bowtie_\epsilon V$, referred to as a distance similarity self-join.

The result of distance similarity searches can be used to optimize the performance of other algorithms. The Density-Based Spatial Clustering of Applications with Noise (DBSCAN) clustering algorithm creates an unspecified number of clusters, where they each have a minimum density. This density requires points to have a minimum number of points (*minPts*) within a distance ϵ to form a cluster, or to be within ϵ of a point in an already existing cluster. The DBSCAN algorithm mostly uses Euclidean distances to evaluate if points are within ϵ, and the overall algorithm can be optimized by first recording all pairs of points that are within ϵ of each other using a parallel distance similarity search algorithm for example, and by then executing the clustering algorithms to form the clusters [46]. The k-Nearest Neighbors (k-NN) algorithm is a particular case of distance similarity search, as it only records the k closest points to each query point. Thus, similarly to the DBSCAN optimization mentioned before, when recording all pairs of points that are within ϵ, we can record only the k closest to each query point to compute a k-NN search [42].

Within the literature, we see that the majority of proposed works use a common optimization, which consists of indexing the points before computing distance similarity searches, using the assumption that certain points are more likely to be similar (i.e., within ϵ) than others. This method greatly reduces the overall complexity of distance similarity searches, as it avoids the computation of the Euclidean distance between points that may not be within ϵ of each other. The method consists of two steps: *search* and *refine*. Given a query point q, we *search* the index for candidate points that are likely to be within ϵ from q. The candidate points are then *refined* by computing the Euclidean distance between the query point and each of the candidates. Only those points that are within ϵ are stored in the result set, R.

Depending on the dataset characteristics, query points from the same dataset may have a greatly varying number of candidate points to refine, and therefore greatly varying workload. Many different indexes exist with different characteristics, and two major types can be identified: hierarchical indexes using trees, and non-hierarchical indexes using grids, and they usually target either low or high dimensional datasets. Indeed, as d increases, the cost to search an index exponentially increases as well. This is because the volume of the data space increases exponentially with d, which is known as the *curse of dimensionality* problem [15, 50, 51]. Consequently, to achieve good performance, novel methods have been designed to address this problem [29, 44, 65, 82]. Additionally, indexes are typically designed for a particular architecture in mind, e.g., either the CPU [13, 14, 16, 17, 21, 33, 48, 57, 87] or the GPU [22, 29, 42, 44, 45, 46, 65, 92], but rarely both [42, 59, 88].

Note that we discuss distance similarity searches from the literature in greater detail in the subsequent chapters.

2.3 Task and Data Parallelism

Designing parallel algorithms involves splitting the work between computing resources. Among the existing methods to split the total work to compute, we identify the two principal methods as follows:

- *Task-parallelism*: the algorithm consists of multiple tasks to compute, and each of these tasks is assigned to either the CPU or GPU architecture. Depending on the nature of the task and its workload, it may be more suited to be executed on the CPU or the GPU. In this case, the architectural differences might be the most important criteria determining the performance of an algorithm.

- *Data-parallelism*: the algorithm consists of an overall single task, which should be executed on several data items. Hence, each piece of data to process is assigned to a processing unit, a CPU core or a GPU for example. Depending on the corresponding workload of the data to process, it may be more suited to assign it to the CPU or the GPU. In this case, the computational throughput of the processors might be the most important factor to determine the amount of work processors should be assigned.

2.4 CPU Architecture

The CPU is the central component of all modern computers. The performance of CPUs largely depends on particular architectural features such as:

- The number of cores, which allows the CPU to execute multiple tasks at the same time, whether it is multiple different tasks or a single task that is parallelized. Because the CPU cores are independent of each other, the CPU is proficient at both task and data-parallel approaches. Today, the number of cores CPUs have can range from just a few to about a hundred. Compute-bound algorithms are typically limited by the number of CPU cores; therefore, increased parallelism in an algorithm should also increase overall performance.

- The system's main memory (RAM), is used to store the data and instructions to process. The main memory can be very large, up to several terabytes, but is also relatively slow and has a high latency, particularly compared to on-chip memory accesses such as cache memory (detailed below). Memory-bound algorithms are typically limited by the bandwidth and latency of the main memory, particularly when several cores need to simultaneously share the resource.

- The size and the levels of cache, which provide high bandwidth and low latency access to certain data, typically that are accessed the most. Instructions and data are fetched from the RAM into the cache for faster accesses, and data are typically flushed out of cache when not used recently. Common processor architectures have three hierarchical levels of on-chip cache, L1 to L3, where each level increases the size but also the latency. L1 and L2 are usually private to a core, while L3 is shared among multiple cores on a CPU.

- The Instruction-Level Parallelism (ILP) hardware allows a CPU's thread of execution to process several instructions at the same time, such as an addition and a multiplication on independent data elements.

- The Single Instruction Multiple Data (SIMD) paradigm, allows CPU cores to execute an instruction on vectorized data. Vectorized instructions are usually made through automatic compiler optimizations or the explicit use of intrinsic CPU functions [55].

Overall, while CPU architectures saw great advancements in the past, nowadays, architecture changes are very limited and are plateauing, where most changes focus on increasing the number of cores, larger cache memory, and newer generations of RAM with greater bandwidth.

2.5 GPU Architecture

The GPU is an accelerator that was originally designed for graphics rendering purposes, which is nowadays extensively used for scientific computation due to its high theoretical peak performance. Some of its main features are:

- GPUs typically have several thousands of cores, and are therefore the architecture of choice for highly parallel algorithms. These general-purpose cores follow the Single Instruction Multiple Thread (SIMT) paradigm, where the same instruction is executed simultaneously on a set of 32 threads, called a warp (which is defined in hardware). Divergent executions between the threads of a warp are computed sequentially, thus reducing performance [23, 78, 79]. Furthermore, because all 32 threads must end their execution at the same time, in the case of an algorithm with irregular workloads and workload imbalance between the threads of a warp, those with less work will idle, waiting for the threads with more work to complete their execution. In the case of Nvidia GPUs, these cores are referred to as CUDA cores, and these cores are grouped into Streaming Multiprocessors (SMs). In the case of other GPU manufacturers, AMD for example refers to the cores as Stream Processors, which are grouped into Compute Units.

- The GPU is equipped with onboard memory, referred to as global memory, and is off-chip relative to the processor. This memory has a high aggregated maximum bandwidth (up to several terabytes per second [76]), which is achievable when threads access coalesced data but also suffers from high latency. In addition, each SM has its own on-chip shared memory, which has a much higher bandwidth and lower latency than global memory. Thus, the shared memory is typically used to manually page frequently accessed data from the global memory for faster retrievals.

- With many threads and a high latency memory, most of the performance of the GPU comes from a rapid hardware context switching to hide the memory access latency by executing threads that have their instructions and data ready to process.

- The overall representation of the threads is as follows: 32 threads are grouped in a warp (defined in hardware), multiple warps constitute a block, and a block is executed on one GPU multiprocessor. Hence, the warps in the same block can exchange data through the on-chip shared memory. Finally, the blocks are grouped into a grid, which thus contains all the threads, each having their own unique identifier. Note that while the concepts of blocks and grids are an abstraction, the warps are defined in the hardware of the GPU.

- We consider here the case where the GPU is discrete, having its own dedicated memory. The CPU (the host) and the GPU (the device) can communicate through one of several interconnects available, such as the PCIe or NVLink [80]. CPU-GPU communications are a known bottleneck due to the relatively low bandwidth of the PCIe interconnect. This issue tends to be alleviated with more recent revisions of this PCIe interconnect, or with the use of the NVLink technology [62, 80].

Because the GPU is nevertheless only an accelerator, the CPU is still required to perform some basic yet mandatory tasks. These tasks typically include allocating memory in the GPU's global memory, copying data from the main memory to the global memory and vice versa, invoking GPU kernels, among other minor tasks. Thus, GPU algorithms also utilize the CPU in several ways. However, we consider that these tasks alone make the CPU underutilized compared to its overall capabilities, and GPU algorithms are thus not fully using the compute resources of the system.

2.5.1 GPU Tensor Cores

Contrary to CPU architectures, GPU architectures have rapidly evolved between each generation. Among the recent features added to GPU architectures, Tensor Cores (TCs) are one of the most important. TCs are a type of Application-Specific Integrated Circuit (ASIC), solely designed to compute Matrix Multiply-Accumulate (MMA) operations. Using four matrices A, B, C and D, TCs can only compute $D = A \times B + C$, and where C and D may be equal and point to the same memory location. Due to their high specificity, TCs have a significantly higher computational throughput when computing MMA operations than CUDA cores.

In the literature, most of the proposed algorithms that use TCs are related to machine learning, linear algebra, or other closely related fields. However, despite their high specificity (only one computation is possible), TCs can be leveraged in any algorithm that uses or can use MMA operations. While this may require significant algorithm modifications, leveraging TCs can significantly improve the performance of an algorithm [4, 27, 56, 63, 67].

TCs can be used through different methods, including high-level libraries such as cuBLAS or CUTLASS [74, 75]. We focus in this dissertation on the low-level Warp Matrix Multiply-Accumulate (WMMA) API, available through the CUDA API [8, 77, 78]. Using the WMMA API offers the most control over the code and over how TCs are used and can be integrated into an already existing kernel contrary to the cuBLAS and CUTLASS functions. However, the API has several constraints. Typically, the WMMA API offers only a limited combination of compute precisions and matrix sizes, which we give in Table 2.1.

Table 2.1: Available standard floating-point precisions and matrix sizes of matrices A, B, C and D using the WMMA API [78].

Precision		Matrix Size		
A, B	C, D	A	B	C, D
FP16	FP16/FP32	16×16	16×16	16×16
FP16	FP16/FP32	32×16	16×8	32×8
FP16	FP16/FP32	8×16	16×32	8×32
FP64	FP64	8×4	4×8	8×8

Note that in hardware, each TC computes on a 4×4 matrix. Hence, because the available matrices are larger than 4×4, several TCs may be used to compute an MMA operation on matrices larger than 4×4. Additionally, the WMMA API provides a class `fragment` to represent the matrices, which are stored across the registers assigned to 32 threads (a warp). Furthermore, the API yields access to a set of functions to use on the fragments:

- *load_matrix_sync*: copies data from a memory starting address to a fragment, and accepts a stride to use to copy consecutive rows/columns of the fragment.

- *store_matrix_sync*: copies data from a fragment to a memory address, also accepting a stride between consecutive rows/columns.

- *fill_fragment*: fill a fragment with a single specified value.

- *mma_sync*: computes an MMA operation between several fragments.

All these functions must be called by all threads of a warp (i.e., 32 threads). Furthermore, while the individual elements of fragments can be individually accessed, the WMMA API does not specify the order in which the elements are stored in the fragment. Hence, unless an instruction should be applied to all the elements of the fragment, we can not alter them independently.

2.6 Comparison of CPU and GPU Architectures

We presented in Sections 2.4 and 2.5 the main features of the CPU and GPU architectures, respectively, and we summarize in this section the main differences. For this comparison, we select the model of CPU and GPU that equip the Leonardo supercomputer, as it is the most efficient supercomputer from the TOP500 list with Nvidia GPUs [89]. The CPUs are Intel Xeon Platinum 8358 [52], and the GPUs are Nvidia A100 [73].

Table 2.2: Comparison of the Intel Xeon Platinum 8358 CPU [52, 53] and Nvidia A100 GPU [73] architectures and performance, both processors equipping the Leonardo supercomputer [89].

Feature	CPU	GPU
Number of cores	32	6912 FP32/INT32, 3456 FP64, 432 TCs
Main/global memory size	Up to 6 TB	40/80 GB
Main/global memory bandwidth (GB/s)	204.8	1555/2039
Preferred parallelism	Task or Data	Data
Peak throughput (TFLOPS)	0.972 [53]	78 FP16, 19.5 FP32, 9.7 FP64, 312 FP16 TCs, 19.5 FP64 TCs

We summarize in Table 2.2 the difference between the Intel Xeon Platinum 8358 CPU [52] and the Nvidia A100 GPU [73]. We choose these processors as they equip the supercomputer equipped with Nvidia GPUs with the highest peak throughput from the TOP500 list [89] at this time. As explained in the previous sections (Sections 2.4 and 2.5), we can see that the main characteristics of the CPU and GPU yield to a significantly greater peak throughput for the GPU than for the CPU, while also having a much higher peak memory bandwidth.

2.7 Heterogeneous Computing

As previously described, computers, and particularly clusters, are increasingly more heterogeneous as they are increasingly equipped with CPU(s) and GPU(s) [89]. Hence, heterogeneous algorithms leveraging both architectures are essential to maximize performance and resource utilization. However, there are few reports in the literature of algorithms that leverage both CPUs and GPUs concurrently, and where both architectures compute a similar amount of work relative to their respective performance, such as the workload between both architectures is relatively balanced. In these proposed works, the CPU is most of the time alleviating some of the work from the GPU, but remains mostly underutilized [59, 88]. Regardless of the approach to split the work among processors (Section 2.3), a heterogeneous computing algorithm leveraging a CPU and a GPU is likely to face several issues, including the following:

- *Load-balancing*: assign work to the processors so their workload is balanced relative to their own computational capabilities. In the case where the processors have the same capabilities, then the workload can be assigned equivalently (e.g., assigning work to multiple CPU cores). When using heterogeneous architectures, because of the architectural differences and vastly different computational throughput, load-balancing is not as straightforward, and several workload partitioning strategies exist, such as dynamic partitioning vs. static partitioning.

- *Hardware heterogeneity*: as described above, CPUs and GPUs have very different architectures, and algorithms designed for either architecture are unlikely to perform as well on the other architecture. Hence, simply designing a GPU program in the same spirit as a CPU program is most certainly going to perform relatively poorly, and optimizing specific aspects of the algorithm to accommodate both architectures at the same time can be non-trivial.

Overall, while heterogeneous computing might be more challenging than designing an algorithm for a single architecture, it is the only way of fully utilizing the computational resources of a machine, and to attempt to achieve better performance than CPU-only and GPU-only algorithms.

Chapter 3

Load Imbalance Mitigation Optimizations for GPU-Accelerated Similarity Joins

This chapter consists of the peer-reviewed article appearing in the 2019 IEEE High-Performance Big Data, Deep Learning, and Cloud Computing Workshop (HPBDC) [35], as part of the International Parallel and Distributed Processing Symposium conference. *This paper won the best paper award of the workshop.* Note that while we present the entire published paper for clarity, part of the contributions were conducted outside the scope of this dissertation, and are reported in my masters thesis [34]. The most important insights of the work were conducted as part of this dissertation, and we estimate that over 50% of the material is unique to the dissertation.

Abstract

The distance similarity self-join is widely used in database applications and is defined as joining a table on itself using a distance predicate. The similarity self-join is often used in spatial applications and is a building block of other algorithms, such as those used for data analysis. In this paper, we propose several new optimizations mitigating load imbalance of a GPU-accelerated self-join algorithm. The data-dependent nature of the self-join makes the algorithm potentially unsuitable for the GPU's architecture, due to variance in workloads assigned to threads. Consequently, we propose a method that reduces load imbalance and

subsequent thread divergence between threads executing in a warp by considering the total workload assigned to each thread, and forcing the GPU's hardware scheduler to group threads with similar workloads within the same warp. Also, by leveraging a grid-based index, we propose a new balanced computational pattern for both reducing the number of distance calculations and the workload variance between threads. Moreover, we exploit additional parallelism by increasing the workload granularity to further improve computational throughput and workload balance within warps. Our solution achieves a speedup of up to 9.7× and 1.6× on average against another GPU algorithm, and up to 10.7× with an average of 2.5× against a CPU state-of-the-art parallel algorithm.

3.1 Introduction

Given a dataset, the distance similarity self-join performs a range query around each point in the dataset to find each point's neighbors within a distance ϵ. The operation is used to find objects that share common properties. In databases, the self-join joins a table on itself, and this operation has been used in various fields as a building block to several algorithms such as data cleaning [24], near-duplicate detection [91], document similarity [12], or clustering algorithms [19, 20, 32].

Given a dataset, D, finding all the points within ϵ of a query point q is an expensive operation. A brute force implementation consisting of two nested loops has a time complexity of $O(|D|^2)$ [31]. Hence, several optimizations have been proposed to improve the performance of the distance similarity self-join, including indexing schemes, which allow pruning the search space, thus avoiding comparisons between a point and every other point in the dataset. There are hierarchical and non-hierarchical indexing schemes. The hierarchical indexing schemes are typically implemented as trees [14, 16, 17, 48, 58, 59, 60], while non-hierarchical structures are often implemented as grids [21, 43, 46, 57]. However, as the dimensionality of the dataset increases, the efficiency of these indexes to prune the search largely diminishes. This effect is called the curse of dimensionality [15, 50, 51].

Several databases operations have already been optimized to be able to benefit from the most recent high-performance computing technologies such as the GPUs [10, 43, 46, 58, 59, 60, 65, 66]. Indeed, their architecture and their high-throughput make them particularly efficient at processing large volumes of data, and it is no exception for the self-join operation [43, 46, 65]. Due to their Single Instruction Multiple Threads (SIMT) architecture, GPUs are very efficient for parallel computations as threads are executed in groups called warps in CUDA terminology [78] (which we use throughout this paper). Threads in a warp are executed in parallel and in lock-step. Branching during the execution is resolved by serializing the execution of threads and their execution pathways, thus causing a loss of parallel efficiency [78]. The GPU's architecture presents several challenges concerning the distance similarity self-join. Among these issues, the total result set size may exceed global memory capacity, particularly in lower dimensions as shown in [43]. Moreover, as state-of-the-art CPU algorithms have been extensively studied, the use of a GPU may not yield a performance advantage over some of the most efficient parallel CPU algorithms.

The similarity self-join is an irregular application where each point in the dataset may not have the same number of distance calculations, depending on the data distribution. Because of the GPU's SIMT architecture, some threads with a higher workload will execute more work and thus, longer than some other threads within the same warp which will have idle periods. This issue may lead to intra-warp load imbalance and therefore decrease the throughput of the self-join. In this paper, we aim at mitigating the load imbalance of the threads within a warp, as well as between different warps. In particular, this paper makes the following contributions:

- We increase the workload granularity of each range query by assigning multiple threads to compute the distance between a query point and its potential neighbors. Having more threads computing the same query point reduces the workload imbalance within a warp, as these threads will share the same workload.
- A new cell access pattern that reduces the number of redundant point comparisons.

This pattern ensures that each thread assigned to a query point compares to the same number of cells, thus potentially reducing load imbalance relative to previous work.

- We reduce intra-warp load imbalance by packing points with similar workloads, quantified by the number of point comparisons, into the same warp. We order each warp in non-increasing order using their quantified total amount of work. Then, we force the GPU's hardware scheduler to execute these warps in this non-increasing order, from most to least work. This ensures that the GPU cores are saturated with work by reducing the time that individual threads within a warp are idle, additionally reducing load imbalance between GPU cores at the end of the computation.
- As we solve the load balancing issue within individual kernels, we effectively improve the throughput of the self-join. As GPU's cores are active for more of the computation compared to previous work, we make better use of them.

This paper is organized as follows: Section 3.2 outlines background and related work, Section 3.3 presents the proposed solutions to mitigate load imbalance, Section 3.4 evaluates the performance of our optimizations, and Section 3.5 concludes the work and discusses future work directions.

3.2 Background

In this section, we first formalize the problem of the distance similarity self-join, and then present related work. We also present the work we leverage and optimize.

3.2.1 Problem Statement

Let D be a dataset containing points in n dimensions. For every query point q_i, $i = 1, \dots, |D|$, we denote its coordinates as x_j, $j = 1, \dots, n$. Thus, $q_1(x_1)$ represents the coordinate in the first dimension of the point q_1. The distance similarity self-join will return all points $q_i \in D$ that are within the Euclidean distance ϵ of each other.

Let p, q be two points in D, where p is within ϵ of q if $dist(p, q) \leq \epsilon$, where $dist(p, q) = \sqrt{\sum_{a=1}^{n}(p(x_a) - q(x_a))^2}$. We elect to use the Euclidean distance as it is a common metric in low dimensionality and to facilitate a direct performance comparison with other implementations of the similarity self-join. All our processing occurs in-memory, and to avoid exceeding the GPU's global memory capacity, we use a batching scheme to incrementally compute the self-join result set across several batches. Moreover, we define a range query as the computation of the ϵ-neighborhood of a query point.

3.2.2 Related Work

Many studies have presented improvements to self-join performance. A common property between all these works is the use of the search-and-refine strategy to improve performance. The search part leverages a data indexing to bound the search space to candidate points that may be within ϵ of a query point. The refine part consists of computing the distance between the query point and its candidate points to only consider those within ϵ. We present an overview of related work regarding indexing, other similarity joins, and range queries.

3.2.2.1 Data Indexing

Based on the distance threshold, indexing allows for retrieving only those points that are likely to be within ϵ of a query point. Index efficacy is based on data properties. Therefore, an index designed for low dimensional data is unlikely to be suited for high dimensional data, and vice versa. Note that all index structures' efficacy degrades in higher dimensionality, which is why we focus on the low-dimensionality similarity self-join. Moreover, some of the indexing methods are suited to the CPU and are not directly applicable to the GPU without a significant performance loss due to several factors. As stated in Section 3.1, hierarchical indexes such as trees and non-hierarchical structures such as grids are the two most prominent solutions for efficient indexing and pruning of the data, thus improving performance. We detail these two methods as follows.

Tree-based Indexing: Index-trees are widely used data structures for the similarity self-join and are particularly suited to those running on a CPU. These indexes are typically constructed based on the data distribution. The R-Tree [48] uses bounding boxes to partition data that are stored in the leaves of the tree. However, the use of bounding boxes as in [48] makes it not well suited to higher dimensions, as data are likely to be assigned to more than a single disjoint partition. Thus, some of the inner bounding boxes will have duplicate data due to their overlapping, leading to both an increase in memory usage and traversal time, as the number of paths traversed increases. Thus, the R*-Tree as proposed in [14] optimizes the area, the margin and the overlapping of these bounding boxes, while the X-Tree [17] improves this overlapping in higher dimensions. The k-d tree [16] is a binary tree organizing points in a k-dimensional space whose nodes are one of the two partitions of the space stored in their parent node. The k-d tree performs reasonably well in higher dimensions as there is no data duplication, each point is in a single disjoint partition.

Although the use of trees is not particularly suited to an efficient use on the GPU due to their many branch instructions and the recursive calls required to traverse them, several works address these issues to improve their efficiency on the GPU. Authors from [60] convert the recursive calls of the R-Tree into sequential accesses, while in [58] they optimize the R-Tree to execute on the GPU by also avoiding recursive calls, as well as improving the irregular memory accesses. In [59], a hybrid R-Tree using both the CPU and the GPU is proposed where the tree is traversed on the CPU, which then sends the data contained in the leaves to the GPU. The CPU performs the tree traversal, which has an irregular execution pattern, and the GPU performs the filtering of points in the leaf nodes. Hence, this exploits each architecture's relative strengths.

Grid-based Indexing: Statically partitioned grid-based indexing consists of partitioning the data into a grid of cells with length ϵ in each dimension. This data structure allows constraining the search of an ϵ-neighborhood of a query point to only its surrounding cells as proposed for a CPU implementation in [21]. Hence, in n dimensions, each point needs to consider up to 3^n cells. Super-EGO, as advanced in [57], is considered a state-of-the-art CPU parallel algorithm and uses a non-materialized grid indexing. In [43], the authors propose a grid indexing targeting the GPU, that only indexes non-empty cells. By doing so, the data structure memory footprint remains very low with a $O(|D|)$ space complexity. We thus compare our optimizations to Super-EGO [57] and this GPU solution [43].

3.2.2.2 Range Queries and Similarity Joins

The Locality-Sensitivity Hashing (LSH) algorithms, such as the E^2LSH algorithm [6, 7, 83], provides an estimated result of the nearest-neighbor search and can be used as an estimated distance similarity search [83] method working well in very high dimensions. However, we do not consider E^2LSH in this paper as it targets high-dimensional data and computes an estimated result, whereas we target lower dimensions and an exact result. The LSS algorithm as proposed in [65] also computes an estimation of the similarity-join by leveraging the use of a GPU and by using space-filling curves, turning the similarity join problem into a sort-and-search problem, which are two very efficient operations on the GPU. This technique creates the curves by sorting on the GPU; then each query point performs an interval search to find candidate points, efficiently pruning the search.

3.2.3 Overview of Leveraged Previous Work

We address load imbalance in the GPU self-join work of Gowanlock & Karsin [43]. We give a brief overview of their work, but refer the reader to additional details in [43]. In contrast to previous work, we focus on several kernel optimizations to mitigate load imbalance.

3.2.3.1 GPU Grid Index

We reuse the ϵ grid indexing for the GPU as proposed in [43]. This method uses several arrays to efficiently store the data into cells of length ϵ. When performing a range query around a query point, this technique bounds the search to only adjacent grid cells. Moreover, as their method only indexes the non-empty cells, the memory footprint is very low, having a space complexity of $O(|D|)$, making it well-suited to the GPU's limited memory capacity. Moreover, each thread performs the same bounded search by accessing neighboring cells, thus reducing the divergence of the threads within the same warp. We reuse their index, which we denote as I.

3.2.3.2 Batching Scheme

Depending on the dataset and the value of ϵ, the self-join may generate a result set size exceeding the GPU's global memory capacity. In [43, 46], the authors advance a solution to prevent buffer overflow. Through a sequence of batches consisting of multiple kernel executions, they compute the self-join while not exceeding the GPU's global memory capacity by transferring partial results back to the host. Thus, with a combination of multiple kernel invocations, pinned memory, and GPU streams, they avoid all global memory buffer overflow and are also able to hide data transfer overhead by overlapping them with kernel executions. This technique samples a percentage of the dataset to estimate the total result set size, yielding the number of batches that need to be executed. In this work, we sample 1% of the entire dataset. The number of batches, *nbBatches*, is determined by the desired maximum result set size for each kernel execution of size b_s. In this paper, we fix this value $b_s = 10^8$ and we use 3 streams. Thus, when using 3 streams, the total pinned memory buffer size is 3×10^8. We define a *batch* as one kernel invocation of the self-join which returns a partial result set, where several batches are needed to compute the entire self-join result. Moreover, we define D_l as the data points assigned to the *batch* l, where $l = 0, 1, \ldots, nbBatches - 1$.

Figure 3.1 represents an example of how threads are assigned across multiple *batches*. We use for this example a dataset D of 12 points and 3 *batches*, $nbBatches = 3$. Therefore, each *batch* has 4 points and thus 4 threads, which are strided across the dataset. Hence, the query point $q_i \in D$ is computed by the *batch* $l = (i - 1)\ mod\ nbBatches$.

D	q_1	q_2	q_3	q_4	q_5	q_6	q_7	q_8	q_9	q_{10}	q_{11}	q_{12}
	l_0	l_1	l_2	l_0	l_1	l_2	l_0	l_1	l_2	l_0	l_1	l_2

Figure 3.1: Example representation of the thread assignment across multiple *batches*. $|D| = 12$, split across 3 *batches* $numbatch = 3$ with 4 points each. l corresponds to the *batch* computing the query point $q_i \in D$, where $i = 1, \ldots, 12$ and $l = (i - 1)\ mod\ nbBatch$.

3.2.3.3 GPU Kernel

The GPUCalcGlobal kernel, as advanced by Gowanlock & Karsin in [43], is the foundation of several of our optimizations. This kernel computes the ϵ-neighborhood of each point in a dataset D, and where each query point in D is computed by a single thread on the GPU. Thus, $|D|$ threads are used. This kernel is given in Algorithm 1, and is a slightly modified version from [43] to use an index I instead of defining each index component described in [43]. The kernel first retrieves the thread's global id (determined by the block's id, the block's size and the thread id within its block), then returns if the thread's global id is larger than the size of the *batch*, as it uses one thread per query point (lines 2 and 3). On line 5, the thread gets its point corresponding to its global id, as well as the neighboring cells on line 6. Then, for each neighboring cell that was found (line 7), we retrieve the list of points contained in the cell (line 8). Afterward, for each candidate point from the neighboring cell (line 9), we compute the distance between this candidate point and the query point (line 10). If the candidate point is within ϵ (line 11), then we add the pair of both points to the result set (line 12). Finally, when a query point has completed its distance calculations, the kernel returns (line 13).

3.2.3.4 Unidirectional Comparison (Unicomp)

Gowanlock & Karsin [43] have advanced a cell access pattern designed to eliminate any duplicate calculations between the points for datasets in any dimension. As the Euclidean distance is a symmetric function ($dist(p, q) = dist(q, p)$), they can add both pairs of points to the result set, with only one distance calculation. This cell access pattern, although pre-

Algorithm 1 GPUCalcGlobal Kernel (GPU) from [43]

```
procedure GPUCALCGLOBAL($D_l, I, \epsilon$)
    gid ← getGlobalId()
    if gid $\geq |D_l|$ then return
    resultSet ← $\emptyset$
    point ← $D_l$[gid]
    adjCells ← getNeighboringCells(gid)
    for cell $\in$ adjCells do
        pointsArr ← getPoints(cell)
        for candidatePoint $\in$ pointsArr do
            result ← calcDistance(point, candidatePoint, $\epsilon$)
            if result $\neq \emptyset$ then
                atomic: gpuResultSet ← gpuResultSet $\cup$ result
    return
```

senting improved response time in most of their experimental evaluations, seems to present an uneven workload balance between threads. For example, in two dimensions, a point may compare with points in up to eight adjacent cells, whereas some points may not compare to any adjacent cell. The implementation of this UNICOMP cell access pattern is given in two dimensions in Algorithm 2, as described in [43]. UNICOMP relies on the odd multidimensional coordinates of the cells to establish an access pattern. It takes as input the query point *point*, C_a its multidimensional coordinates, $filteredRngs$ the range of the non-empty cells in each dimension, and B the array of non-empty cells. If the first coordinate of a cell is odd (line 2), then this cell is a part of the pattern. The algorithm iterates over the first dimension (line 3), and if the explored cell does not have the same first index as the origin cell (line 4), then the linear id of the neighboring cell is calculated (line 5). If this linear id corresponds to a non-empty cell (line 6), the origin point is compared to the points of the neighboring cell (line 7). Lines 8 to 14 are used to iterate over the second dimension. As this example is for two dimensions indexing, an additional loop is needed for each additional dimension. The comments "Green arrows" and "Red arrows" (respectively lines 2 and 8) refer to the arrows represented in Figure 3.2.

Figure 3.2 represents the cell access pattern of Unicomp in two dimensions. The arrows

Algorithm 2 The Unicomp cell access pattern in 2 dimensions (GPU) from [43]

```
procedure UNICOMP2D($point, C_a, filteredRngs, B$)
    if $C_a$.x is odd then                                        ▷ Green arrows
        for x ∈ $filteredRngs[1]$ do
            if x ≠ $C_a$.x then
                linearID ← getLinearCoord(x, $C_a$.y)
                if linearID ∈ $B$ then
                    ComparePoints($point$, linearID)
    if $C_a$.y is odd then                                          ▷ Red arrows
        for x ∈ $filteredRngs[1]$ do
            for y ∈ $filteredRngs[2]$ do
                if y ≠ $C_a$.y then
                    linearID ← getLinearCoord(x, y)
                    if linearID ∈ $B$ then
                        ComparePoints($point$, linearID)
    return
```

represent the neighboring cells to compare to, while the numbers in the cells quantify the number of neighboring cells that are compared.

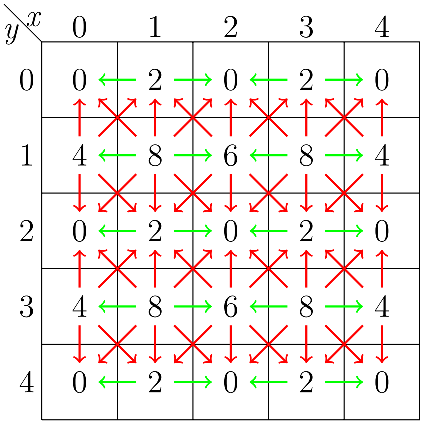


Figure 3.2: UNICOMP cell access pattern in two dimensions. The numbers represent the number of neighboring cells the origin cell is going to compare, and the arrows indicate these neighboring cells. While green arrows indicate an odd x index, red arrows are for an odd y index.

3.3 Mitigating Load Imbalance

The SIMT architecture of the GPU makes it well-suited for highly data-parallel applications. Threads are executed in groups of 32 called warps [78]. Due to hardware limitations (e.g., the number of available registers), only a limited number of warps can be executed concurrently on the GPU. In the case of the distance similarity self-join, the workload is dependent on the data distribution, therefore potentially disparate within threads of the same warp. For example, in a real world dataset, some points will have few neighbors, and some will have many neighbors, potentially spanning several orders of magnitude. In this situation where threads of the same warp have both points from a dense region and points from a sparse region, some of these threads will be idle for a longer amount of time than others. While the threads computing the points from a dense region of the dataset are still active, this prevents the execution of a new warp.

Figure 3.3 is an example representation of the possible workload imbalance we might face within a warp when using the original GPUCalcGlobal kernel described in Section 3.2.3.3. Due to intra-warp workload imbalance, some of the threads will be idle while some others will be computing, thus reducing the GPU's resources usage efficiency.

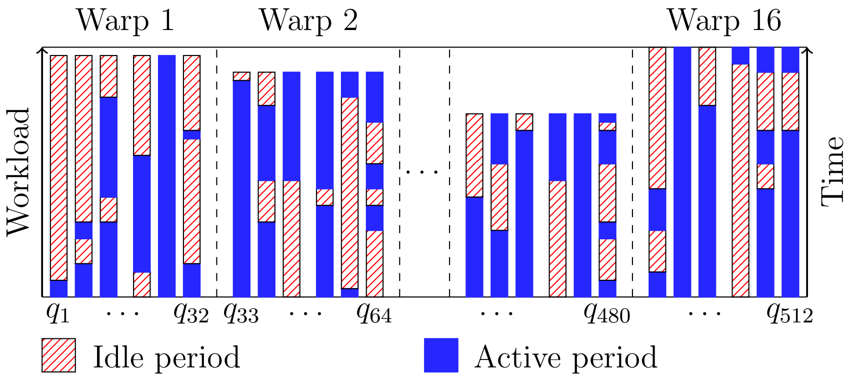


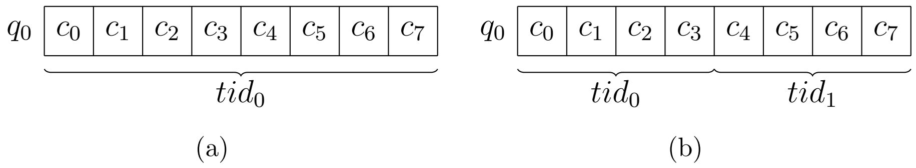
Figure 3.3: Example representation of the workload across a dataset, with q_1 to q_{32} query points in the first warp, q_{33} to q_{64} query points in the second warp, and q_{481} to q_{512} in the last warp, assuming $|D| = 512$. This represents the potential workload imbalance of the original GPUCalcGlobal kernel.

3.3.1 Increasing the Granularity of each Range Query

The GPU kernel advanced by Gowanlock & Karsin [43] uses a single GPU thread per query point. Thus, a single thread is computing every distance calculation between its point and all the neighboring points. Depending on the properties of the data, some query points may have many distance calculations to compute, and therefore large amounts of work. Consequently, if one thread is assigned to compute all of the distance calculations, then at the end of a kernel execution, some of the GPU cores will be idle. Therefore, we can increase the granularity of the filtering task by assigning multiple threads to each query point for computing the distances. This reduces the amount of idle resources at the end of the computation.

An optimization is to use multiple threads per query point, so each thread is computing a fraction of the distance calculations of its assigned query point. This will reduce the workload of each thread, and thus reduce the time needed to find the neighbors of the query point. Moreover, by assigning the same workload to each thread of a query point, and as the number of thread within a warp is fixed, using this optimization will reduce the intra-warp load imbalance. However, increasing the total number of threads implies a larger number of warps to schedule. We define k as the number of threads assigned to a single query point.

Figure 3.4 represents how we assign threads to the candidate points c_i of a query point q_j. In the present case, we use as an example a query point q_0 with a neighboring cell containing eight points: c_0 to c_7, and $k = 2$. Figure 3.4 (a) represents how all candidate points are assigned to the single thread as in the GPUCalcGlobal kernel, while Figure 3.4 (b) represents the candidate points being assigned to the two different threads. For descriptive purpose, we assume k evenly divides the number of candidate points.



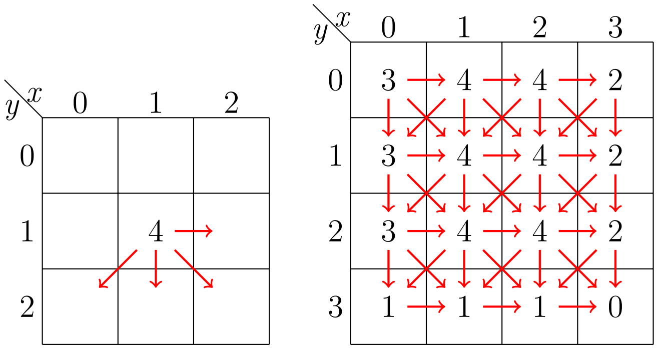
(a)

(b)

Figure 3.4: (a) Original assignment of a thread to candidate points, as in [43], (b) Assignment of threads to candidate points when increasing the distance calculation granularity, with $k = 2$ (even case). q_0 is a query point with eight candidate points ($\{c_0,\ldots,c_7\}$). tid_i designates the local thread id of the query point, where $i = 0, \ldots, k-1$.

3.3.2 Cell Access Pattern: Linear ID Unidirectional Comparison

We propose Linear ID Unidirectional Comparison (Lid-Unicomp) as an optimization to the Unicomp cell access pattern advanced in [43]. While Unicomp relies on extensive conditional statements to determine whether points of a cell need to compare to neighboring cells, Lid-Unicomp reuses the fact that with the grid indexing we use, non-empty cells have a unique linear id computed from the cell's coordinates in n dimensions. The principle of this new cell access pattern is thus to compute the distance with the points from every neighboring cell that has a higher linear id than the origin cell. The cell access pattern of this method is represented in Figure 3.5. As we observe if we compare Figure 3.2 to Figure 3.5, when using the Lid-Unicomp pattern, every inner cell will compare to the same number of neighboring cells. Depending on the data distribution, this might greatly improve the workload balance over Unicomp because it has some cells comparing to every neighboring cell, and some cells comparing to none.



(a)

(b)

Figure 3.5: Overview of the Lid-Unicomp pattern in 2-D. The numbers represent the number of neighboring cells the origin cell is going to compare to, and the arrows indicate these neighboring cells. (a) represents the cell access pattern on its own in 2-D, (b) represents its application on a 2-D grid.

Algorithm 3 gives the implementation of the Lid-Unicomp cell access pattern. For each neighboring cell (line 3), if the linear id of the neighboring cell is greater than the linear id of the origin cell (line 5), then following the Lid-Unicomp cell access pattern, this cell needs to be evaluated (line 6).

In comparison with the Unicomp cell access pattern (Section 3.2.3.4), the implementation of the Lid-Unicomp pattern is more straightforward as it relies on a linear id calculation and a single condition, whereas Unicomp relies on an extensive combination of loops and conditions. An advantage of Unicomp is that it stops searching as soon as an iterated multidimensional coordinate is even (Algorithm 2, lines 2 and 8), while Lid-Unicomp checks the linear id of all the non-empty adjacent cells. Consequently, Unicomp does not need to iterate over all adjacent cells in comparison to Lid-Unicomp. Thus, Unicomp may outperform Lid-Unicomp, depending on several data-dependent factors.

Algorithm 3 Lid-Unicomp cell access pattern implementation (GPU)

```
procedure LidUnicomp($q, originCell, \epsilon$)
    originId ← linearId($originCell$)
    for c ∈ getNeighborCells($originCell$) do
        neighborId ← linearId(c)
        if originId < neighborId then
            evaluateNeighborCell($q$, c)
```

3.3.3 Local and Global Load Balancing: Sorting by Workload

Consider two threads t_0 and t_1, where t_0 is assigned a query point in a sparse region (q_0 in Figure 3.6), and t_1 is assigned a query point in a dense region (q_1 in Figure 3.6). t_0 will perform 14 distance calculations, and t_1 will perform 45 distance calculations. If these threads are within the same warp, t_1 will have much more work than t_0, and t_0 will be idle for a significant amount of time as it waits for t_1.

To reduce the amount of time that threads are idle, a solution is to sort the points by their workload (number of point comparisons), such that each warp be assigned threads with similar workloads in comparison to an unbalanced workload as in Figure 3.3. This sorting is achieved by computing the number of distance calculations of each non-empty cell, i.e., retrieving their number of neighbors, and assigning points from the cell with the greatest workload at the beginning of a new array denoted as D'. Furthermore, as a consequence of the batching scheme (Section 3.2.3.2), the data points assigned to each *batch* D_l have a similar

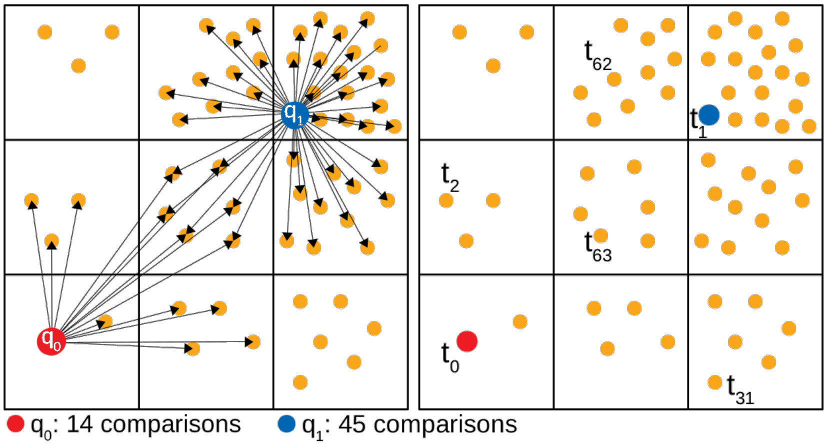


Figure 3.6: Illustration of the load imbalance between query points and therefore between threads, where q_0 and q_1 are two query points and t_0 and t_1 two threads processing the query points q_0 and q_1 respectively.

total workload due to accessing the data in a strided manner (Figure 3.1). SortByWL is applied to each *batch* D'_l and not D'; therefore warps will not be strictly assigned points from most workload to least work in the scope of the entire dataset. However, this ensures that each *batch* does not overflow the result set buffer.

3.3.4 Forcing the Warp Execution Order using a Work Queue

SortByWL does not entirely obviate load imbalance as threads within the same warp still have different workloads due to the stride of the threads across the dataset as presented in Section 3.2.3.2. Moreover, the hardware scheduler may not execute the warps from most workload to least work, as the scheduler still has control over the execution order of warps. To obviate these issues, we propose using a priority queue. While previous work implementing a queue on the GPU exists [90], they use a distributed queue with dynamic load balancing where threads can retrieve or give work to other threads. Moreover, they use their threads for the entire computation duration, making it unsuitable for our work due to our batching scheme. Therefore, we do not use the queue from [90] and extend our SortByWL optimization using a queue that is persistent across all kernel invocations. Thus, complementary to SortByWL which outputs a sorted array of the points, we use a global counter to indicate the equivalent of the head of a queue. Each thread increments this counter through an atomic operation that assigns data points to threads. By using this optimization, we expect our workload to be nearly identical between the threads of the same warp, as the example represented in Figure 3.7, where the idle periods of the threads are significantly reduced in comparison to Figure 3.3.

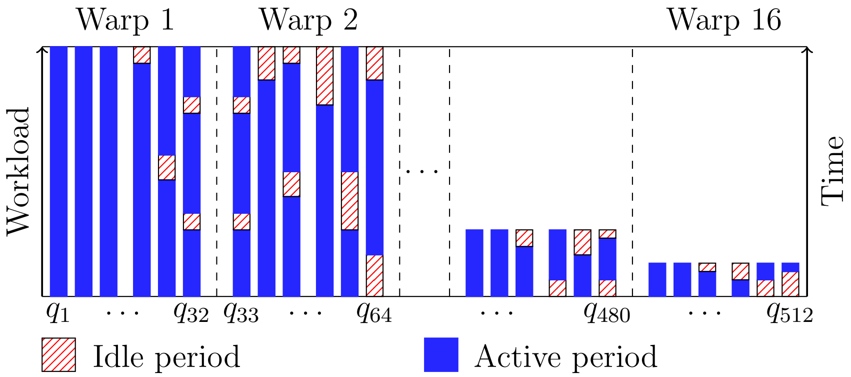
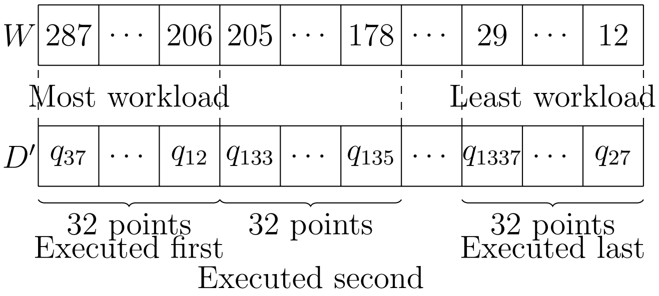


Figure 3.7: Representation of balancing the workload between the threads within the same warp. We use for this example a dataset $D = 512$.

Figure 3.8 represents the functioning of our WorkQueue optimization, where D' is our dataset sorted by workload and W indicates the total workload of a query point. The workload is quantified as the number of distance calculations a query point will perform to refine its candidate set.



W	287	⋯	206	205	⋯	178	⋯	29	⋯	12
D'	q_{37}	⋯	q_{12}	q_{133}	⋯	q_{135}	⋯	q_{1337}	⋯	q_{27}

Figure 3.8: Representation of the points' execution order when using the WorkQueue optimization. D' is the sorted dataset, and W gives the workload of each query point. The first 32 points with the most workload will be executed at the beginning within the same warp, while the last 32 points, with the least workload, will be executed at the very end.

Unlike SortByWL, we consider the entire dataset D' (as sorted by workload) when executing *batches*, and do not employ adding points to D_l across *batches* in a strided manner. This ensures that each warp has the smallest possible variance in workloads (point comparisons). However, this leads to large variance in result set sizes across *batches*, which the strided D_l *batches* were designed to avoid. Consequently, in the WorkQueue optimization, we slightly modify the batching scheme (Section 3.2.3.2), and instead of sampling the entire dataset to estimate the total result set size, we sample the first 1% of D' (without striding), which yields a much larger estimated total result set size. This ensures that our first *batch* does not overflow the result set buffer; however, we execute more batches than when using GPUCalcGlobal or SortByWL.

Finally, when we use the WorkQueue in combination with a $k > 1$, we use cooperative groups introduced with CUDA 9.0 [49]. We thus create groups of size k where only the first thread increments the global counter and then shuffles the returned result to the other threads of the cooperative group.

3.4 Experimental Evaluation

3.4.1 Datasets

To evaluate our proposed solutions, we select several datasets presenting different characteristics such as the dimensionality and size. We consider datasets synthetically generated with a uniform, and an exponential distribution with $\lambda = 40$, each composed of two million points between two and six dimensions. We use these both distributions as they present opposite workloads, and therefore to outline the impact of our optimizations. For the real world datasets, we use the *SW-* datasets [69] with 1.86M and 5.16M points, both in two and three dimensions representing the latitude and longitude of the objects in two dimensions, including the total number of electrons in the ionosphere as the third dimension. Moreover, we select 50 million points from the *Gaia* catalog [2] in two dimensions. For the synthetic datasets, we denote *Expo-* as exponentially distributed datasets and *Unif-* as uniformly distributed datasets. The summary of these datasets is given in Table 3.1. We omit the three, four and five-dimensional data of our synthetic datasets for the intermediate plots (Figures 3.9, 3.10 and 3.11), as two and six dimensions bound the performance.

Table 3.1: Summary of the different datasets used for the experimental evaluation. $|D|$ denotes the number of points and n the dimensionality.

Dataset	$\|D\|$	n	Dataset	$\|D\|$	n	Dataset	$\|D\|$	n
Unif2D2M	2M	2	*Expo2D2M*	2M	2	*SW2DA*	1.86M	2
Unif3D2M	2M	3	*Expo3D2M*	2M	3	*SW2DB*	5.16M	2
Unif4D2M	2M	4	*Expo4D2M*	2M	4	*SW3DA*	1.86M	3
Unif5D2M	2M	5	*Expo5D2M*	2M	5	*SW3DB*	5.16M	3
Unif6D2M	2M	6	*Expo6D2M*	2M	6	*Gaia*	50M	2

3.4.2 Methodology

We use a platform composed of 2×Intel E5-2620v4@2.10 GHz for a total of 16 cores, coupled with 128 GiB of RAM and an Nvidia Quadro P100 (16 GiB of HBM2 global memory). The GPU code is written in CUDA, while the C/C++ host code is compiled with the GNU compiler with the O3 flag. The response times do not include the index construction time because we do not optimize index construction in the implementations that we compare to. All other components of the algorithm are included in the response time.

In all GPU experiments, we use 256 threads per block, and each data point is represented as a 64-bit floating point. The parallel CPU Super-EGO experiments include the time to EGO-sort and join, and use 32-bit floating points and run using 16 threads across 16 physical cores, yielding the best configuration on our platform. Table 3.2 outlines the optimizations and notation used in the experimental evaluation.

We average the response times over three trials, while we profile on only three *batches* as each *batch* has nearly identical performance characteristics. Although we retrieve several different metrics through the Nvidia Profiler [70], we choose only to report the warp execution efficiency in this paper, as it is the most relevant metric among those we have collected regarding the performance of our optimizations.

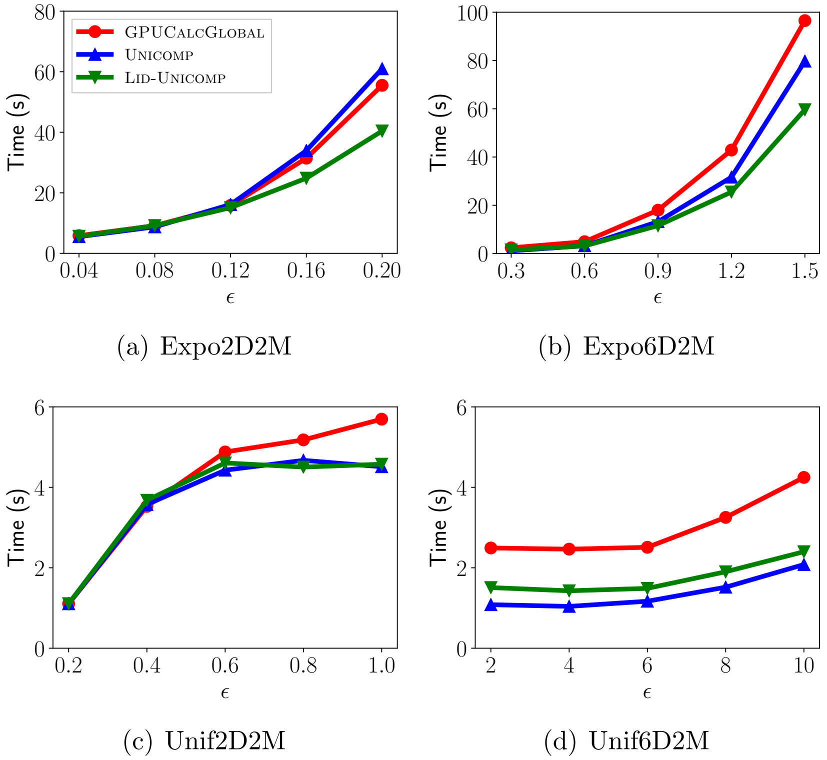
Table 3.2: Optimizations and notation used throughout the evaluation.

Notation	**Description**
GPUCalcGlobal	Original GPU kernel [43] we compare to.
Unicomp	Original cell access pattern [43] we compare to.
Super-EGO	State-of-the-art CPU parallel algorithm [57] that we compare to.
Lid-Unicomp	Cell access pattern advanced in Section 3.3.2.
SortByWL	Sorting by workload optimization (Section 3.3.3).
WorkQueue	Work-queue optimization (Section 3.3.4).
k	Number of thread per query point (Section 3.3.1).

3.4.3 Results

Impact of the New Cell Access Pattern: Here, we evaluate the response time of the Lid-Unicomp cell access pattern optimization (Section 3.3.2) in several scenarios and compare it to the response time of the GPUCalcGlobal and Unicomp kernels, the two solutions we aim to improve. Figure 3.9 plots the response time vs. ϵ of the GPUCalcGlobal kernel, and the Unicomp and Lid-Unicomp cell access patterns on our uniformly and exponentially distributed datasets, in two and six dimensions. We observe that Unicomp has a lower response time than GPUCalcGlobal, excepting the *Expo2D2M* dataset when $\epsilon > 0.12$ (Figure 3.9 (a)). Moreover, our solution, Lid-Unicomp, improves the performance of the self-join in most cases, except on the *Unif6D2M* dataset (Figure 3.9 (d)).

To understand these results, we profile the execution of these three configurations, on the *Expo2D2M*, *Expo6D2M*, *Unif2D2M*, and *Unif6D2M* with $\epsilon = 0.2, 1.2, 1.0, 8.0$, respectively. We report the results in Table 3.3. The warp execution efficiency is the average percentage of active threads in each executed warp. We choose this metric as having a high warp execution efficiency means that only a few threads are idle during the execution of each warp. The warp execution efficiency between Unicomp and Lid-Unicomp is correlated to the response time. In most cases, as the warp execution efficiency is higher for Lid-Unicomp than Unicomp, the response time is lower, with an exception for the *Unif6D2M* dataset (Figure 3.9 (d)). Regarding the GPUCalcGlobal kernel, despite a higher warp execution efficiency than the Unicomp or Lid-Unicomp optimizations, its response time is higher.



(a) Expo2D2M (b) Expo6D2M

(c) Unif2D2M (d) Unif6D2M

Figure 3.9: Response times of the Lid-Unicomp cell access pattern, versus the GPUCalcGlobal kernel and the Unicomp cell access pattern on our synthetic datasets. The legend in (a) is used across all subfigures, and we set $k = 1$.

This is because both cell access patterns reduce the number of distance calculations by a factor of roughly two, thus improving the response time. Thus, the proposed Lid-Unicomp optimization may be more efficient than the previous Unicomp cell access pattern due to its more evenly distributed work across threads. Moreover, the low warp execution efficiency on the exponentially distributed datasets may reflect intra-warp workload imbalance.

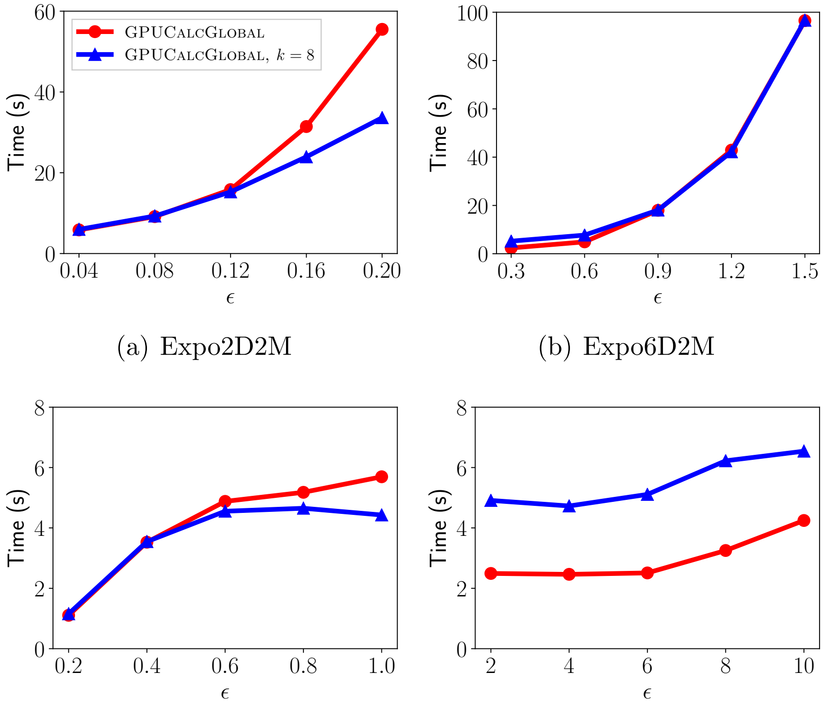
Table 3.3: Warp execution efficiency (WEE) of the GPUCalcGlobal kernel as well as the Unicomp and Lid-Unicomp cell access patterns over our synthetic datasets and for specific values of ϵ. The time corresponds to that in Figure 3.9.

		GPUCalcGlobal		**Unicomp**		**Lid-Unicomp**	
Dataset	ϵ	WEE(%)	Time(s)	WEE(%)	Time(s)	WEE(%)	Time(s)
Expo2D2M	0.2	26.5	55.5	13.2	60.9	18.3	40.4
Expo6D2M	1.2	15.2	42.9	7.8	31.6	10.0	25.5
Unif2D2M	1.0	75.4	5.7	48.94	4.5	69.1	4.6
Unif6D2M	8.0	51.3	3.3	19.25	2.1	40.9	2.4

Impact of Assigning Multiple Threads to Each Query Point: We now focus on the performance of increasing the thread granularity, specifically by using eight threads per point ($k = 8$). We compare this optimization to the GPUCalcGlobal kernel, which uses only one thread ($k = 1$), and use the same datasets as for the Lid-Unicomp performance evaluation. Having $k > 1$ reduces the workload of each thread, by reducing the number of distance calculations each of them has to compute. Moreover, this also reduces the workload variance within a warp, as the threads computing the same query point will share the same total workload. Figure 3.10 plots the response time of the GPUCalcGlobal kernel when $k = 1$ and when $k = 8$ on our synthetic datasets. While the *Expo2D2M* dataset (Figure 3.10 (a)) greatly benefits from the increased granularity when $\epsilon \geq 0.12$, the response time is not impacted on the *Expo6D2M* dataset (Figure 3.10 (b)) and performs even worse when $\epsilon \leq 0.9$. Regarding the uniformly distributed datasets, while *Unif2D2M* presents a lower response time when having $k = 8$ and $\epsilon \geq 0.4$ (Figure 3.10 (c)), the GPUCalcGlobal kernel with $k = 1$ performs better on the *Unif2D2M* dataset (Figure 3.10 (d)). Therefore, having a low workload as it is the case for lower values of ϵ seems not to be suited to an increase of the workload granularity, although it does not especially degrade performance. The exception is on the *Unif6D2M* dataset, which performs better when $k = 1$ for every ϵ values.

Table 3.4 shows the warp execution efficiency and the response time of the GPUCalcGlobal kernel when $k = 1$ and $k = 8$. We observe that having more threads greatly increases the warp execution efficiency, particularly for the exponentially distributed datasets. This observation is reflected in the response time, which is lower for our selected values of ϵ. However, although the warp execution efficiency is always higher when $k = 8$, the response time of this configuration is higher on the *Unif6D2M* dataset than for $k = 1$. We leave investigating this behavior for future work.

Impact of Reordering the Points by Workload and Forcing Warp Execution Order: We evaluate the performance of our SortByWL and WorkQueue optimiza-



(a) Expo2D2M

(b) Expo6D2M

(c) Unif2D2M

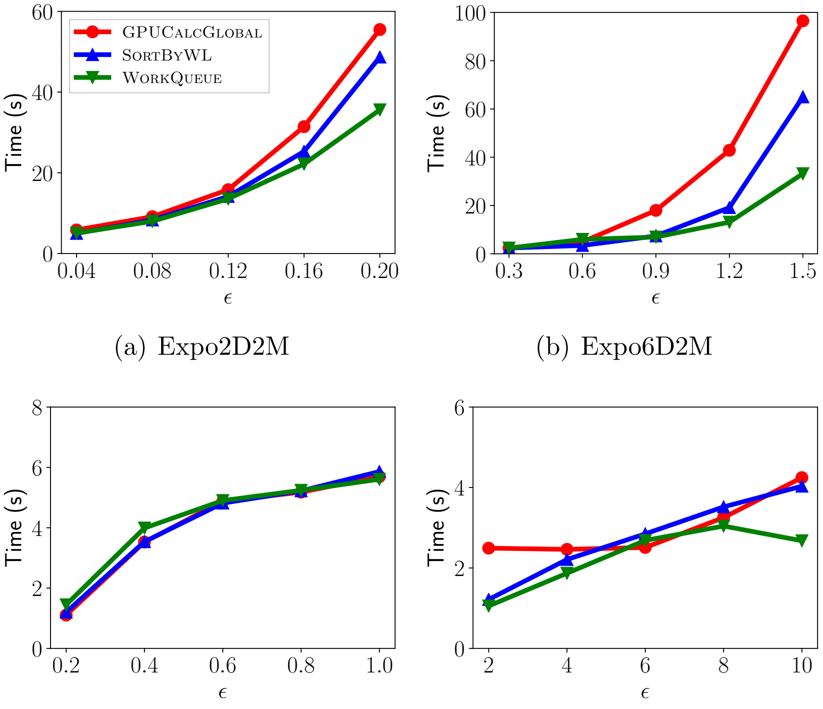
(d) Unif6D2M

Figure 3.10: Response time of the increase of the granularity when $k = 8$ versus $k = 1$ for the GPUCalcGlobal kernel on our synthetic datasets. The legend in (a) is used across all subfigures.

Table 3.4: Warp execution efficiency (WEE) of the GPUCalcGlobal with $k = 1$ and when $k = 8$ on synthetic datasets and for specific values of ϵ. The time corresponds to that in Figure 3.10.

		GPUCalcGlobal		**GPUCalcGlobal, $k = 8$**	
Dataset	ϵ	WEE (%)	Time (s)	WEE (%)	Time (s)
Expo2D2M	0.2	26.5	55.5	40.8	33.6
Expo6D2M	1.2	15.2	42.9	39.27	42.2
Unif2D2M	1.0	75.4	5.7	80.3	4.4
Unif6D2M	8.0	51.3	3.3	60.9	6.2

tions, compared to GPUCalcGlobal. Figure 3.11 plots the response time vs. ϵ of the GPUCalcGlobal kernel, and our SortByWL and WorkQueue optimizations on our uniformly and exponentially distributed datasets, in two and six dimensions. Observing the exponentially distributed datasets in two and six dimensions (Figures 3.11 (a)-(b)), we see an improvement in the response time, particularly for higher values of ϵ. For smaller values of ϵ, the workload variance between points is reduced, thus decreasing the impact of packing the points based on their workload. Moreover, even without controlling the execution workflow when using the SortByWL optimization, it performs better than GPUCalcGlobal in every case on the exponentially distributed datasets. Nevertheless, the WorkQueue thus seems to be very effective, especially on datasets with significant variance of workload between points, as expected. However, sorting the points based on their workload does not present any significant gain when datasets are uniformly distributed as every point have a similar workload, unlike exponentially distributed datasets. We observe this on Figures 3.11 (c)-(d)) where neither SortByWL or WorkQueue significantly outperform GPUCalcGlobal.



(a) Expo2D2M

(b) Expo6D2M

(c) Unif2D2M

(d) Unif6D2M

Figure 3.11: Response time of the SortByWL and WorkQueue optimizations against the GPUCalcGlobal kernel on our synthetic datasets. The legend in (a) is used across all subfigures, and we set $k = 1$.

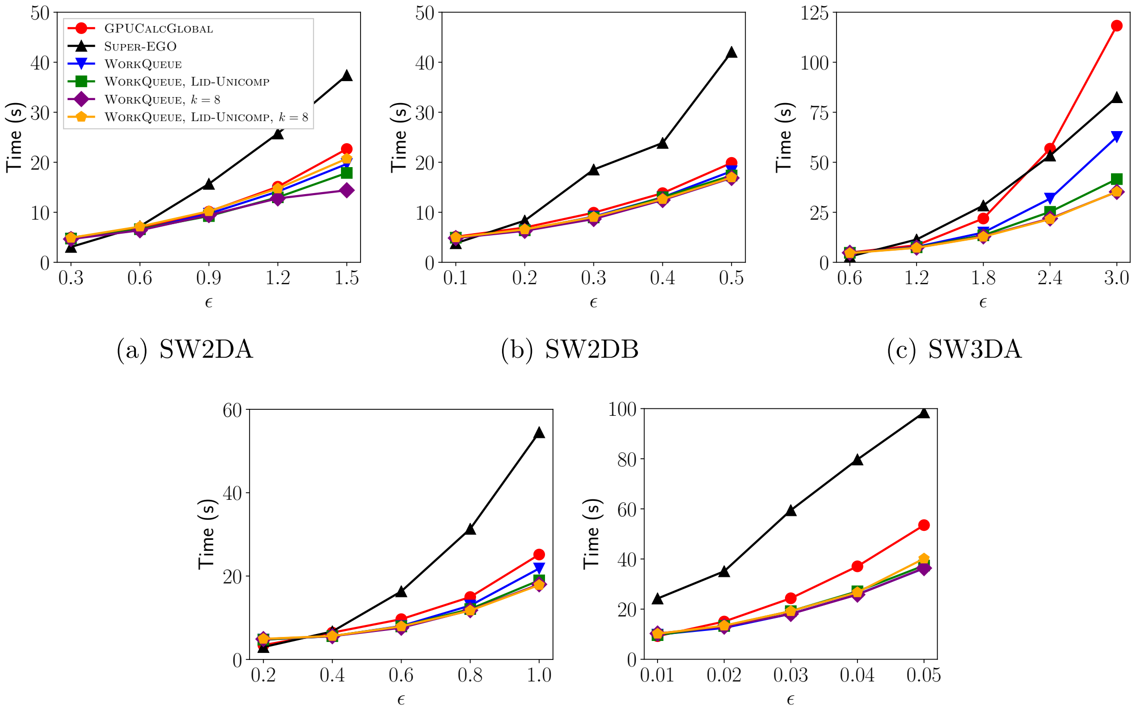
In Table 3.5, we observe that the warp execution efficiency is much higher when using WorkQueue. Moreover, an increase of the warp execution efficiency results in a decrease of the response time, excepting the *Unif2D2M* dataset. The WorkQueue presents both the highest warp execution efficiency and the lowest response time on our selected configurations. Therefore, our strategy of packing warps with similar workloads and forcing the hardware scheduler to execute warps in order clearly improves performance.

Table 3.5: Warp execution efficiency (WEE) of the GPUCalcGlobal kernel, as well as the SortByWL and WorkQueue optimizations on our synthetic datasets and for specific values of ϵ. The time corresponds to that in Figure 3.11.

		GPUCalcGlobal		**SortByWL**		**WorkQueue**	
Dataset	ϵ	WEE(%)	Time(s)	WEE(%)	Time(s)	WEE(%)	Time(s)
Expo2D2M	0.2	26.5	55.5	74.6	48.7	83.2	35.6
Expo6D2M	1.2	15.2	42.9	71.4	19.1	95.6	13.1
Unif2D2M	1.0	75.4	5.7	75.4	5.9	83.1	5.6
Unif6D2M	8.0	51.3	3.3	48.2	3.5	48.4	3.0

Combination of Approaches on Real World Datasets: Figure 3.12 plots the response time vs. ϵ using a combination of our optimizations, including WorkQueue with Lid-Unicomp and $k = 8$. This combination of optimizations outperforms GPUCalcGlobal and Super-EGO across nearly all experimental scenarios. In particular, our optimizations are the most effective on the largest workloads (large datasets and ϵ). The performance on the real world datasets is limited to $n = 3$ dimensions. Thus, the performance of our optimizations typically converge across the datasets because the workloads are low at $n \leq 3$ dimensions, but we will show that on higher dimensionality (Figure 3.13), the combination of all optimizations will yield larger performance gains (e.g., the *Expo6D2M* dataset, Figure 3.11 (b)).

Table 3.6 shows the warp execution efficiency and the total response time for selected values of ϵ from Figure 3.12. All of our solutions present a better warp execution efficiency and overall response time than GPUCalcGlobal, which indicates that warp execution efficiency is a good metric for assessing load imbalance. Due to the high warp execution efficiency observed across all of our optimizations in Table 3.6, we believe that further optimizations to the GPUCalcGlobal kernel are not likely to lead to significant performance



(a) SW2DA

(b) SW2DB

(c) SW3DA

(d) SW3DB

(e) Gaia

Figure 3.12: Response time vs. ϵ on real world datasets of the WorkQueue optimization, the WorkQueue combined with the Lid-Unicomp pattern, the WorkQueue with $k = 8$, and both combined to the WorkQueue compared to the GPUCalcGlobal kernel and the Super-EGO CPU parallel algorithm. The legend in the subfigure (a) is used across all subfigures.

gains. However, new algorithmic designs may improve performance.

3.5 Discussion and Conclusion

The self-join has data-dependent performance behavior and irregular instructions that make the problem challenging to solve efficiently on the GPU. Depending on the data distribution, the self-join leads to load imbalance within each warp, which limits the GPU's throughput. Consequently, we have advanced several optimizations that address load imbalance. We propose a cell access pattern that avoids duplicate computation. In contrast to previous work, this allows each point in the dataset to be compared to the same number of adjacent cells. We increase the granularity of each range query by assigning each query

Table 3.6: Warp execution efficiency (WEE) of the GPUCalcGlobal, the WorkQueue, and the WorkQueue combined with Lid-Unicomp and $k = 8$ on our real world datasets and for specific values of ϵ. The time corresponds to that in Figure 3.12.

		GPUCalcGlobal		**WorkQueue, Lid-Unicomp**		**WorkQueue, $k = 8$ Lid-Unicomp**	
Dataset	ϵ	WEE(%)	Time(s)	WEE(%)	Time(s)	WEE(%)	Time(s)
SW2DA	1.2	55.2	15.1	89.1	13.0	80.7	12.5
SW2DB	0.4	54.2	13.8	83.0	13.0	79.7	12.6
SW3DA	2.4	33.7	56.8	93.4	25.2	83.2	21.6
SW3DB	0.8	40.8	14.9	87.1	12.1	82.5	11.7
Gaia	0.04	64.1	37.1	80.3	27.1	78.3	26.7

point multiple threads for performing the distance calculations. This reduces the number of varying workloads within each warp. We propose packing warps with threads assigned similar workloads to reduce the load imbalance within each warp. Lastly, we ensure that the GPU's hardware scheduler executes warps in non-increasing order of each warp's assigned work. This reduces inter-warp load imbalance, which ensures that the warps finish their execution at similar times, at the end of the kernel execution.

Figure 3.13 summarizes the performance of our WorkQueue, Lid-Unicomp and $k = 8$ optimizations combined on all datasets. From the figure, we find that using the optimizations outlined in this paper, we are able to significantly improve the performance over (a) a parallel CPU implementation, and (b) previous GPU self-join work. We achieve speedups up to 10.7× over Super-EGO and 9.7× over GPUCalcGlobal, with an overage of 2.5× and 1.6× respectively. This work demonstrates that reducing intra-warp workload imbalance can significantly improve performance, and thus has implications for other algorithms with data-dependent performance characteristics.

Future work directions are outlined as follows. We will apply our optimizations to other applications, especially the WorkQueue, which could be adapted to any self-join indexing structure, contingent upon being able to quantify the workload. We will investigate dynamically grouping *batches* of queries together when using the work queue such that each *batch* yields similar result set sizes. Additionally, we will carry out a more extensive performance

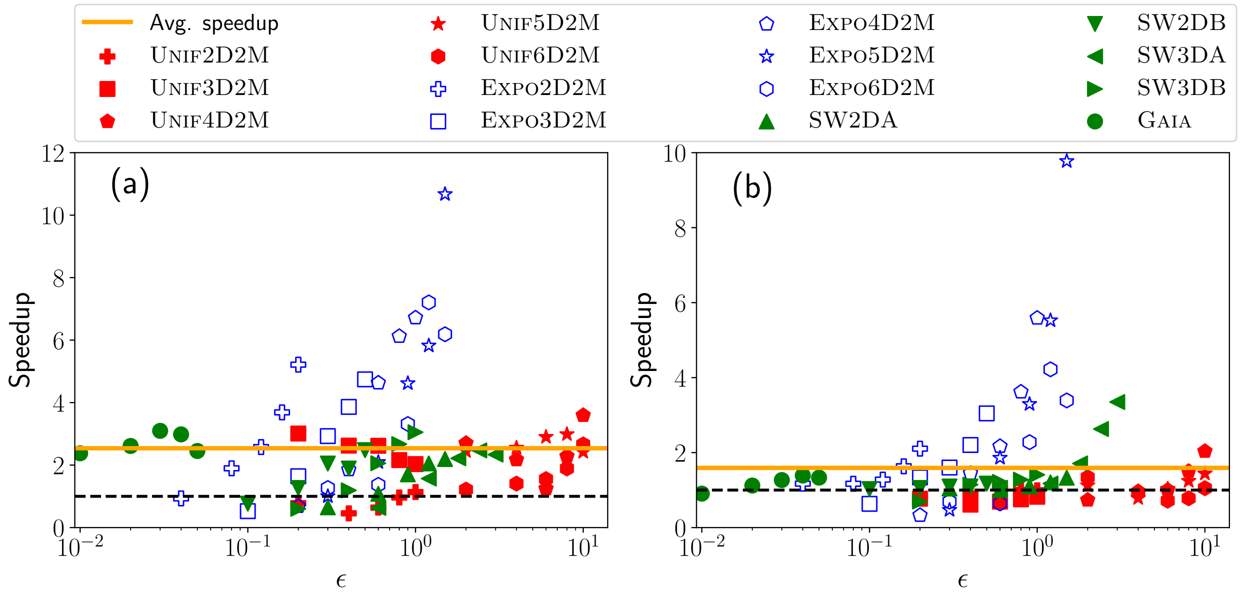


Figure 3.13: Speedup of the WorkQueue combined to Lid-Unicomp and $k = 8$ optimization against the Super-EGO parallel algorithm (a), and over the GPUCalcGlobal kernel (b), on several datasets. ϵ values are plotted on a log scale to observe all data points.

comparison between the proposed cell access pattern and the one proposed by our previous work.

Chapter 4

Heterogeneous CPU-GPU Epsilon Grid Joins: Static and Dynamic Work Partitioning Strategies

This chapter consists of the peer-reviewed article appearing in the proceedings of the Proceedings of the 25^{th} International Conference on Database Systems for Advanced Applications (DASFAA) [36], which was further extended and published in the Data Science and Engineering Journal [37].

Abstract

Given two datasets (or tables) A and B and a search distance ϵ, the distance similarity join, denoted as $A \ltimes_{\epsilon} B$, finds the pairs of points (p_a, p_b), where $p_a \in A$ and $p_b \in B$, and such that the distance between p_a and p_b is $\leq \epsilon$. If $A = B$, then the similarity join is equivalent to a similarity self-join, denoted as $A \bowtie_{\epsilon} A$. We propose in this paper Heterogeneous Epsilon Grid Joins (HEGJoin), a heterogeneous CPU-GPU distance similarity join algorithm. Efficiently partitioning the work between the CPU and the GPU is a challenge. Indeed, the work partitioning strategy needs to consider the different characteristics and computational throughput of the processors (CPU and GPU), as well as the data-dependent nature of the similarity join that accounts in the overall execution time (e.g., the number of queries, their distribution, the dimensionality, etc.). In addition to HEGJoin, we design in this paper a dynamic and two static work partitioning strategies. We also propose a performance model

for each static partitioning strategy to perform the distribution of the work between the processors. We evaluate the performance of all three partitioning methods by considering the execution time and the load imbalance between the CPU and GPU as performance metrics. HEGJoin achieves a speedup of up to 5.46× (3.97×) over the GPU-only (CPU-only) algorithms on our first test platform, and up to 1.97× (12.07×) on our second test platform over the GPU-only (CPU-only) algorithms.

4.1 Introduction

Consider two input datasets A and B, and a distance threshold ϵ. A distance similarity search finds the pairs of points (p_a, p_b), $p_a \in A$ and $p_b \in B$, such that the distance between these two points is $\leq \epsilon$. While any distance function can be used, in the literature, the Euclidean distance is typically employed [20, 21, 22, 35, 57, 65]. These similarity searches are typically computed as a semi-join operation ($A \ltimes_\epsilon B$), where A is a set or table of query points and B a set or table of entries in an index. The particular case where $A = B$ is a self-join (and thus $A \bowtie_\epsilon A$). For simplicity, we examine in this paper the self-join problem. However, we do not explore optimizations exclusive to the self-join. Thus, our optimizations apply to the semi-join case as well. For an input dataset, D, the brute-force self-join solution has a time complexity of $O(|D|^2)$. This complexity decreases when a data indexing method is used to prune the search space. Hence, using an index and the *search-and-refine* strategy, for each query point in D, the *search* of the index generates a set of candidate points that are likely to be within ϵ of the query point, while the *refine* step computes the distance between a query point and its candidate points to produce the final result set.

The indexing methods used for the search-and-refine strategy are often designed for either low [21, 22, 35, 57] or high dimensionality [64, 65, 83]. Due to the *curse of dimensionality* [15, 57], when dimensionality increases, index searches become more exhaustive, and the complexity of the algorithm gradually degrades into a brute-force search. Hence, indexes suited for low dimensional data are likely not to be as efficient when used on higher dimensional data (and vice versa). The curse of dimensionality is thus among the reasons why we only focus here on the low-dimensionality case, rather than any dimensionality: we elect to create an efficient algorithm for the low-dimensional case, rather than a less efficient algorithm that addresses all dimensionalities. Furthermore, while low-dimensional searches are often memory-bound, high-dimensional searches are usually compute-bound, as the cost of a distance calculation increases with dimensionality. In this paper, we focus on low-dimensional searches. Hence, HEGJoin may saturate memory bandwidth, thus potentially negatively impacting performance and parallel scalability of the algorithm, as compared to when fewer processors contend for memory bandwidth.

Graphics Processing Units (GPUs) have been increasingly used for general computational problems, and particularly for improving similarity join performance [22, 65], and with specific data indexing methods that are suited to the GPU's particular Single Instruction Multiple Threads (SIMT) architecture [9, 58, 59, 60, 92]. The proliferation of GPUs is particularly explained by their increased computational throughput and higher memory bandwidth compared to CPUs. However, despite these attractive features, their use in combination with the CPU to perform some part of the computation to further improve database query throughput, such as the distance similarity join, remains underexplored. Thus, we propose in this paper HEGJoin, a heterogeneous CPU-GPU distance similarity search algorithm. Hence, in addition to the CPU performing GPU-supporting tasks (launching kernels, transferring data, etc.), we explicitly use the CPU to compute a fraction of the total number of query points.

As discussed above, the literature concerning heterogeneous CPU-GPU database applications is relatively scarce. Thus, we propose to leverage both the CPU and GPU, and design an efficient algorithm to compute distance similarity searches. There are two major CPU-GPU similarity search algorithm designs, described as follows:

- **Task parallelism:** Assign the CPU and GPU particular tasks to compute, such as *searching* on the CPU and then *refining* on the GPU [59].

- **Data parallelism:** Split the data to compute, and perform both the *search* and *refine* steps on each architecture independently, using different algorithms suited to the strengths of each architecture [41].

In the literature, heterogeneous CPU-GPU similarity search and related range query algorithms focus on a *task-parallel* approach [59, 88]. The *task-parallel* approaches model the problem as a two [59] or three stage pipeline [88], where the CPU is assigned one task, such as searching an index, and the GPU is assigned the task of refining the candidate points [59]. Consequently, as with any pipeline, the throughput is dependent on the slowest stage. Therefore, the drawback of the *task-parallel* approach is that it can leave resources (CPU or GPU) underutilized. In this paper, we focus on the data-parallel approach, which allows us to exploit all available computational resources in the system to maximize query throughput. Since we concurrently use the CPU and GPU, we then need to efficiently partition the work among our processors, i.e., assign to each processor a number of queries to compute so the algorithm achieves good load balancing and thus good performance. To the best of our knowledge, HEGJoin is the first *data-parallel* heterogeneous and concurrent CPU-GPU distance similarity join algorithm.

As our solution is designed for data-parallelism, our work partitioning strategies partition queries from the input dataset. Because HEGJoin is a heterogeneous CPU-GPU algorithm, this is particularly challenging as we need to efficiently distribute the work to accommodate each processor's architectural characteristics. The data-parallel work partitioning can be achieved by different methods: dynamically [41] where the work is assigned to the processors on-demand, or statically [47, 93], where each processor has a fixed amount of work to compute. Statically partitioning the work is challenging, as we need to determine the amount of work to be assigned to the CPU and GPU such that it minimizes load imbalance between the processors. The workload has data-dependent performance characteristics that depend on the number of points, their dimensionality, and their distribution (e.g., underdense vs. overdense regions). Consider partitioning the data using a dynamic approach. In this case,

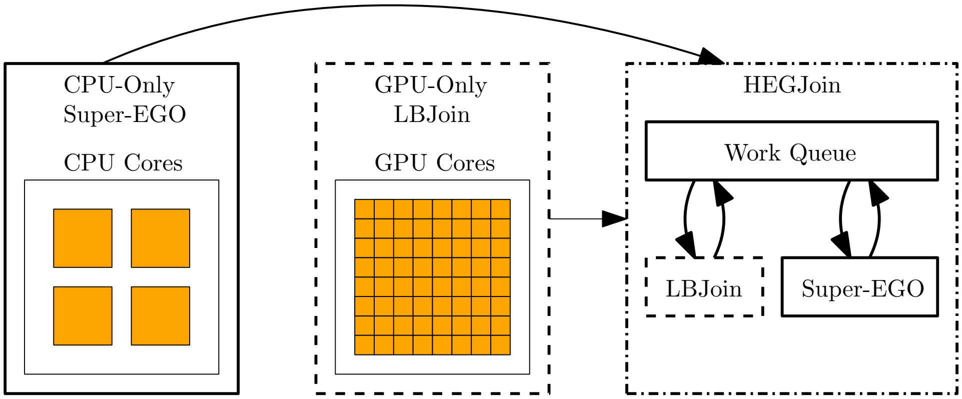


Figure 4.1: Representation of how we combine Super-EGO and LBJoin by using a single work queue to form HEGJoin. When using the static partitioning strategy, the CPU and the GPU would access the work queue only once (at the beginning of the algorithm to retrieve their assigned queries). When using the dynamic partitioning scheme, the CPU and the GPU would iteratively query the work queue for queries to compute until it is empty.

partitioning involves having the pieces of work assigned to the CPU or the GPU, where there is a trade-off between small work units assigned to each processor to achieve good load balancing, and large work units so that the processors reach peak throughput. On the other hand, static partitioning requires accurately estimating the total workload, which is particularly challenging given the data-dependent nature of the work. In contrast, on other problems that have deterministic workloads, the workload can be accurately estimated, and static work partitioning is straightforward [81].

To enable static partitioning, we propose two performance models that quantify the workload based on different metrics that enable the two static partitioning strategies to assign work to the CPU and GPU. Additionally, we propose a dynamic partitioning strategy that is oblivious to the workload. We compare these partitioning strategies to assess their relative strengths and weaknesses, to understand how the characteristics of the workload may affect the performance of HEGJoin, and to ultimately be able to select the partitioning strategy that yields the best performance.

Our algorithm leverages two previously proposed independent works that were shown to be highly efficient: the GPU algorithm (LBJoin) by Gallet and Gowanlock [35] and the CPU algorithm (Super-EGO) by Kalashnikov [57]. However, although we mention above that HEGJoin employs a data-parallel approach, as we leverage two different algorithms (LBJoin and Super-EGO) and a work queue, our algorithm also has task-parallel characteristics. While the output of LBJoin and Super-EGO is identical, the algorithm executed by the CPU is inherently different from the algorithm executed by the GPU. Therefore, HEGJoin uses a mixed parallelism model (a combination of data- and task-parallelism). Figure 4.1 illustrates how LBJoin and Super-EGO work together through the use of a single shared work queue.

By combining the LBJoin and Super-EGO algorithms and using our work partitioning methods, we achieve better performance on most experimental scenarios than CPU-only or GPU-only approaches. Note that, since Super-EGO and LBJoin respective indexing methods are more efficient in lower dimensions, and as most of the related literature works rarely focus on both low and high dimensionality, we choose to focus on low dimensional distance similarity joins. Hence, this paper makes the following contributions:

1. We combine state-of-the-art algorithms for the CPU and GPU to propose a new algorithm, HEGJoin, and which is, to the best of our knowledge, the first data-parallel heterogeneous and concurrent CPU-GPU distance similarity join algorithm.

2. We propose an efficient shared double-ended work queue (deque) to assign query points either to the CPU or to the GPU. Furthermore, we exploit the GPU's high computational throughput by assigning it query points with the highest workload (located at the beginning of the deque), while we assign the query points with the smallest workload to the CPU.

3. We develop three different workload partitioning strategies. The dynamic work partitioning strategy uses the shared deque to assign work to either the CPU or GPU. In this case, there is no fixed boundary on the work that can be assigned to the CPU or the GPU, as it is assigned to processors on-demand. Furthermore, we advance two static work partitioning methods: based on the number of query points, and based on

the total number of candidate points that need to be refined per query point. As with both static strategies the CPU and GPU have a fixed number of queries to compute, if the GPU completes its work before the CPU, it must wait for the CPU to complete its work (and vice versa).

4. We optimize Super-EGO to further improve the performance of HEGJoin. We denote this optimized version of Super-EGO as New-Super-EGO.

5. We evaluate the performance of HEGJoin using seven real-world and ten exponentially distributed synthetic datasets, and using two platforms. We achieve speedups up to 5.46× and 3.97× over the GPU-only and CPU-only algorithms on the first test platform, and speedups up to 1.97× and 12.07× on the second test platform. Furthermore, we achieve an average load imbalance ratio as low as 0.14 when using the dynamic work partitioning strategy on the first platform.

The paper is organized as follows. We begin in Section 4.2 by surveying the literature and presenting an overview of GPU architecture. We then present in Section 4.3 the leveraged algorithms, and we describe HEGJoin and its main features in Section 4.4. We evaluate the performance of HEGJoin and our partitioning methods in Section 4.5, and we finally conclude this paper in Section 4.6.

4.2 Background

4.2.1 Problem Statement

Let D be a dataset in d dimensions. Each point in D is denoted as q_i, where $i = 1, ..., |D|$. We denote the j^{th} coordinate of $q_i \in D$ as $q_i(j)$, where $j = 1, ..., d$. Thus, given a distance threshold ϵ, we define the distance similarity search of a query point q as finding all points in D that are within this distance ϵ to q. We also define a candidate point $c \in D$ as a point whose distance to q is evaluated. Similarly to related work, we use the Euclidean distance. Therefore, the similarity join finds pairs of points ($q \in D$, $c \in D$), such that $dist(q, c) \leq \epsilon$, where $dist(q, c) = \sqrt{\sum_{j=1}^{d}(q(j) - c(j))^2}$. All processing occurs in-memory. While we consider the case where the result set size may exceed the GPU's global memory capacity, we do not consider the case where the result set size may exceed the platform's main memory capacity.

4.2.2 GPU Architecture

We present material related to GPU architecture and use CUDA terminology throughout the paper. Modern GPUs are equipped with a few thousand cores. The global memory bandwidth of the GPU is over an order of magnitude higher than the main memory bandwidth of the CPU (up to 1555 GB/s for the Tesla A100 [73] GPU). However, the GPU's global memory has limited capacity, and the potential for parallelism is dependent on control flow, as threads are executed in groups of 32 (called *warps*) in lock-step. Also, different workloads assigned to threads within the same warp induce idle periods, where some threads are idle while others are computing. The PCI interconnect between the CPU and the GPU is a bottleneck (PCIe-v3 has 32 GiB/s bi-directional bandwidth). For more information on the CUDA programming model and the GPU architecture, we refer the reader to general references on the topic [78, 79].

4.2.3 Related Work

In this section, we outline relevant work regarding the distance similarity join and work partitioning methods between heterogeneous architectures.

4.2.3.1 Data Indexing

Since the similarity join is frequently used as a building block within other algorithms, the literature regarding the optimization of the similarity join is extensive. However, the vast majority of existing literature aims at improving performance using either the CPU or the

GPU, and rarely both. Hence, literature regarding heterogeneous CPU-GPU similarity join optimizations remains relatively scarce. The search-and-refine strategy (Section 4.1) largely relies on the use of data indexing methods, that we describe as follows.

Indexing data structures are used to prune the search space of an indexed input dataset to reduce the number of candidates that may be within ϵ of each query point. Given a query point q and a distance threshold ϵ, indexes find the candidate points that are likely to be within a distance ϵ of q. Also, the majority of the indexes are designed for a specific use, whether they are for low or high dimensional data, for the CPU, for the GPU, or both architectures. We identify different indexing methods, including those designed for the CPU [13, 14, 16, 21, 25, 48, 57, 87], the GPU [9, 45, 58], or both architectures [41, 59, 88]. As our algorithm focuses on the low dimensionality distance similarity search, we focus on presenting indexing methods that are designed for lower dimensions. Since indexes are an essential component of distance similarity searches, identifying the best index for each architecture is critical to achieve good performance, especially when using two different architectures. Furthermore, although our heterogeneous algorithm leverages two previously proposed works [35, 57] that both use a grid index for the CPU and the GPU, we discuss in the following sections several other indexing methods based on trees.

CPU Indexing: In the literature, the majority of indexes designed for the CPU used to index multi-dimensional data are based on trees. The following trees have been designed for range queries and can, therefore, be used for distance similarity searches. The kD-Tree [16] is a binary tree that indexes k-dimensional data by subsequently splitting the search space in two, following an alternation of the k dimensions (in two dimensions for example, split following the x-axis, then the y-axis, then the x-axis, etc.). Hence, each node stores the coordinates of its search space, and splits it between its two child nodes. The Quad Tree [33] is very similar to the kD-Tree, as it consists of a tree whose nodes have four children, and as the search space is subsequently divided into four sub-spaces (instead of two for the kD-Tree). The nodes of the R-Tree [48] consist of bounding boxes to store multi-dimensional objects, which are then stored in the leaf nodes of the tree. In addition to these tree indexes, grids such as the Epsilon Grid Order (EGO) [21, 57] have also been designed for distance similarity joins. We discuss this EGO index that we leverage in Section 4.3.2.

GPU Indexing: Similar to CPU indexes, index-trees have been optimized to address the GPU's SIMT architecture. Kim et al. [58] optimized the R-Tree on the GPU by replacing the recursive accesses inherent to traversing the tree that are not suited to the GPU. They replaced these accesses by sequential accesses, particularly by allowing the search of the tree to jump from a node to its next sibling. Awad et al. [9] improve the efficiency of the B-Tree by using nodes the size of the GPU's cache access size, and by avoiding recursive calls during the tree traversal as well. Furthermore, they assign multiple queries to a warp, with all the threads of the same warp that cooperate to compute one query at a time, thus reducing intra-warp thread divergence. We leverage the GPU grid index proposed by Gowanlock and Karsin [45] and that is designed for distance similarity joins, which we present in Section 4.3.1.1.

CPU-GPU Indexing: Kim and Nam [59] propose an R-Tree designed for range queries that uses task parallelism. The CPU searches the internal nodes of the tree and, when reaching the leaf nodes, sends this partial result to the GPU. The GPU then traverses these leaf nodes, which are stored as a contiguous array in GPU's main memory so the memory accesses are likely coalesced, and refines the candidate objects. In contrast, Gowanlock [41] elects to use two indexes for data parallelism to compute k-NN searches. The CPU uses a kD-Tree [16], while the GPU uses a grid [45]. Hence, both indexes are suited to their respective architecture.

4.2.3.2 Workload Partitioning

As described above, efficiently partitioning the work of parallel algorithms is critical, whether it is based on the tasks to execute (task-parallelism), or the data to compute (data-parallelism). Because the solution we propose in this paper requires data-parallelism, we describe in this section contributions in the literature that propose partitioning schemes for data-parallel algorithms as well.

Efficient work partitioning is usually difficult to achieve since several parameters need to be considered: typically the processors' relative performance (e.g., computational throughput or memory bandwidth), and if the algorithm's workload can be easily determined (usually the case for non data-dependent workloads). On problems exposing a data-dependent workload, such as the distance similarity join or sparse matrix multiplications [68] for example, determining the workload is more challenging than for problems without data-dependent workloads (e.g., regular matrix-matrix multiplications [81]).

Dynamic partitioning solutions [41] present advantages to keeping the processors busy (as they are assigned work until none is available), and do not require knowing the relative performance of the processors beforehand, making it agnostic to platform hardware characteristics. Furthermore, while dynamic work partitioning does not require knowledge about the workload to be functional, it may still be beneficial to determine an overall workload in order to assign work to the most suitable processor.

On the other hand, static partitioning methods [47, 93], if not arbitrary (i.e., a static partitioning of work not based on information related to the processors or the algorithm), requires having accurate knowledge about the relative performance of the processors as well as the workload to achieve good load balancing between the processors. Furthermore, most static partitioning methods are based on models [47, 93], which are made for a specific algorithm and platform. Hence, their solution may be inefficient when used for a different algorithm (which would require a new model) or on a different platform (which would require adapting the model for this new platform).

Gowanlock [41] proposes a dynamic partitioning scheme to compute k-NN searches. Using a work queue, they continuously assign query points to the CPU and the GPU until all the work has been computed. As the overall workload of the algorithm is determined beforehand, they are able to assign more query points and with the highest workloads to the GPU, and the rest to the CPU. The load balancing of the computation is thus managed by the work queue and the dynamic work assignment to the different processors.

Grewe and O'Boyle [47] advance a general static partitioning scheme of applications. Their solution relies on different metrics such as the number of computing operations and their precision, the number of memory operations, the presence of loops, etc., extracted from the code before the computation. Hence, they determine an overall workload for the algorithm and, if the computation is considered to be efficient if executed on both the CPU and GPU, then they estimate a work partitioning using the input data size, and a model they previously developed using the same application and for several fixed static partitioning fractions. Yasuhito Ogata et al. [93] propose a model to statically assign the work of a Fast Fourier Transform (FFT) to the CPU and the GPU. Their solution creates sub-problems of the FFT and, following their model, assign these sub-problems to the most suitable processor. This model is based on parameters such as previously recorded performance, CPU-GPU data transfer rate, memory management on the GPU, matrix transposition performance, and several other factors.

The dynamic work partitioning strategy we propose in this paper, while similar to the one proposed by Gowanlock [41], should be more efficient as the way we determine our workload is more accurate than their solution. Our static work partitioning methods, similarly to other static partitionings [47, 93], also propose a performance model (for each of our static partitioning strategy). However, we outline the importance of determining the overall workload to efficiently partition, by proposing an intuitive solution having rather little knowledge about the workload, and a second method with an accurate knowledge of the workload. The load imbalance between the CPU and GPU would show such importance. For comparative purposes, we expect that our solution with accurate workload knowledge will yield better load balancing than the solution with less knowledge of the workload.

4.3 Leveraged Work

In this section, we present the leveraged works used to design HEGJoin. We use LBJoin [35] for the GPU and Super-EGO [57] for the CPU, which are two state-of-the-art algorithms for their respective platforms, which are publicly available. For greater detail, we encourage the reader to refer to the original papers of Super-EGO [57] and LBJoin [35]. Furthermore, we acknowledge that a CPU distance similarity join algorithm has been proposed in the literature by Perdacher et al. [82] that outperforms Super-EGO at high dimensions. However, their algorithm has comparable performance to Super-EGO in low dimensionality. Therefore, we use Super-EGO and not Perdacher et al. [82] to create HEGJoin, as it is better suited to our low dimensional case.

4.3.1 GPU Algorithm: LBJoin

The GPU component of HEGJoin is based on the GPU kernel proposed by Gallet and Gowanlock [35]. This kernel also uses the grid index and the batching scheme by Gowanlock and Karsin [45]. This work is the best distance similarity join algorithm for low dimensions that uses the GPU (there are similar GPU algorithms but they are designed for range queries, see Section 4.2).

4.3.1.1 Grid Indexing

The grid index presented by Gowanlock and Karsin [45] allows the query points to only search for candidate points within its 3^d adjacent cells (and the query points' own cell), where d is the data dimensionality. This grid is stored in several arrays in the GPU's global memory: *(i)* the first array represents only the non-empty cells to minimize memory usage, *(ii)* the second array stores the cells' linear id and a minimum and maximum indices of the points, *(iii)* the third array corresponds to the position of the points in the dataset and is pointed to by the second array. Candidate points are retrieved by searching the index in global memory, which yields a set of candidates points in the dataset, D. Furthermore, the threads within the same warp access adjacent cells in the same lock-step fashion, thus avoiding thread divergence. Also, note that we modify their work and now construct the index directly on the GPU, which is much faster than constructing it on the CPU as in the original work.

4.3.1.2 Batching Scheme

Computing the ϵ-neighborhood of many query points may yield a very large result set and exceed the GPU's global memory capacity. Therefore, in Gowanlock and Karsin [45], the total execution is split into multiple batches, such that the result set does not exceed global memory capacity.

The number of batches that are executed, n_b, are defined by an estimate of the total result set size, n_e, and a buffer of size n_s, which is stored on the GPU. The authors use a lightweight kernel to compute n_e, based on a sample of D. Thus, they compute $n_b = n_e/n_s$.[1] The buffer size, n_s, can be selected such that the GPU's global memory capacity is not exceeded. The number of query points, n_p^{GPU}, processed per batch (a fraction of $|D|$) are defined by the number of batches as follows: $n_p^{GPU} = |D|/n_b$. Hence, a smaller number of batches will yield a larger number of queries processed per batch.

The total result set is simply the union of the results from each batch. Let R denote the total result set, where $R = \bigcup_{l=1}^{n_b} r_l$, where r_l is the result set of a batch, and where $l = 1, 2, \ldots, n_b$.

The batches are executed in three CUDA *streams*, allowing the overlap of GPU computation and CPU-GPU communication, and other host-side tasks (e.g., memory copies into and out of buffers), which is beneficial for performance.

[1] In this section, for clarity, and without the loss of generality, we describe the batching scheme assuming all values divide evenly.

4.3.1.3 Sort by Workload and Work Queue

The sorting strategy proposed by Gallet and Gowanlock [35] sorts the query points by non-increasing workload. The workload of a query point is determined by the sum of candidate points in its own cell and its 3^d adjacent cells in the grid index. Hence, the grid index is used to retrieve the adjacent cells, and to find the number of points in each of them. This results in a list of query points sorted from most to least workload, which is then used in the work queue to assign work to the GPU's threads. The consequence of sorting by workload and of using this work queue is that threads within the same warp will compute query points with a similar workload, thereby reducing intra-warp load imbalance. This reduction in load imbalance, compared to their GPU reference implementation [35], therefore reduces the overall number of periods where some threads of the warp are idle and some are computing. This yields an overall better response time than when not sorting by workload. This queue is stored on the GPU as an array, and a variable is used to indicate the head of the queue. In this paper, we store this queue on the CPU's main memory to be able to share the work between the CPU and the GPU components of HEGJoin. Furthermore, the sum of the individual workloads of each query point corresponds to the total workload. Since this sorting by workload strategy uses the grid index to compute the workload, it allows for estimating the workload for any input dimensionality and data distribution.

4.3.1.4 GPU Kernel

The GPU kernel [35] makes use of a grid index, the batching scheme, as well as the sorting by workload strategy and the work queue presented above. Moreover, we configure the kernel [35] to use a single GPU thread to process each query point ($|D|$ threads in total). Thus, each thread first retrieves a query point from the work queue using an atomic operation. Then, using the grid index, the threads search for their non-empty neighboring cells corresponding to their query point, and iterate over the found cells. Finally, for each candidate point within these cells, the algorithm computes the distance to the query point and if this distance is $\leq \epsilon$, then the key/value pair made of the query point's id and the candidate point's id is added to the result buffer r of the batch.

4.3.2 CPU Algorithm: Super-EGO

Similarly to our GPU component, the CPU component of HEGJoin is based on the efficient distance similarity join algorithm, Super-EGO, proposed by Kalashnikov [57]. We detail its main features as follows.

4.3.2.1 Dimension Reordering

The principle of this technique is to first compute a histogram of the average distance between the points of the dataset and for each dimension. A dimension with a high average distance between the points means that points are more spread across the search space, and therefore fewer points will join. The goal is to quickly increase the cumulative distance between two points so it reaches ϵ with fewer distance calculations, allowing the algorithm to short-circuit the distance calculation and continue computing the next point.

4.3.2.2 EGO Sort

This sorting strategy sorts the points based on their coordinates in each dimension, divided by ϵ. This puts spatially close points close to each other in memory, and serves as an index to find candidate points when joining two sets of points. This sort was originally introduced by Böhm et al. [21].

4.3.2.3 Join Method

The Super-EGO algorithm takes a set of query points and computes each point's result set as follows. First, in main memory, Super-EGO recursively creates new partitions, until these partitions reach a given size. Next, the join is made by comparing the set of query points to this set of generated partitions of the input dataset, where the partitions that are recursively generated are sets of points spatially co-located to the set of query points. Then, since the points are sorted based on their coordinates and the dimensions have been reordered, two partitions are compared only if their first point is within ϵ from each other. If they are not, then subsequent points will not join either, and the join of the two partitions is aborted.

4.3.2.4 Parallel Algorithm

Super-EGO also adds parallelism to the original EGO algorithm, using pthreads and a producer-consumer scheme to balance the workload between threads. When a new partition is recursively created, if the size of the queue is less than the number of threads (i.e., some threads have no work), the newly created partitions are added to the work queue to be shared among the threads. This ensures that no threads are left without work to compute.

4.4 Heterogeneous CPU-GPU Algorithm: HEGJoin

In this section, we present the major components of our heterogeneous CPU-GPU algorithm, HEGJoin, the different techniques we propose to partition the workload between the CPU and the GPU, as well as improvements made to the work we leverage.

4.4.1 Shared Work Queue

As mentioned in Section 4.3.1, we leverage the work queue stored on the GPU that was proposed by Gallet and Gowanlock [35], which efficiently balances the workload between GPU threads. However, to use the work queue for the CPU and the GPU components of HEGJoin, we must relocate it to the host/CPU to use it with our CPU algorithm component. Because the GPU has a higher computational throughput than the CPU, we assign the query points with the most work to the GPU, and those with the least work to the CPU. Similarly to the shared work queue proposed by Gowanlock [41] for the CPU-GPU

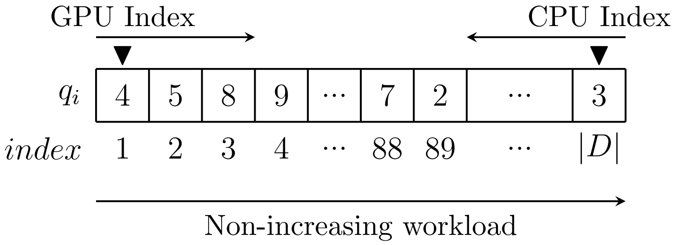


Figure 4.2: Representation of our deque as an array. The numbers q_i are the query points id, the triangles are the starting position of each index, and the arrows above it indicate the indices progression in the deque.

k-NN algorithm, the query points need to be sorted based on their workload, as detailed in Section 4.3.1.3. However, while query points' workload in Gowanlock [41] is characterized by the number of points within each query point's cell, we define here the workload as the number of candidate points within all adjacent cells. Our sorting strategy is more representative of the workload than in Gowanlock [41], as it yields the exact number of candidates that must be filtered for each query point.

Using this queue with the CPU and the GPU requires modifying the original work queue [35] to be a double ended-queue (deque), as well as defining a deque index for each architecture. Since the query points are sorted by workload, we set the GPU's deque index to the beginning of the deque (greatest workload), and to the end of the deque for the CPU's index (smallest workload). Therefore, the GPU's workload is configured to decrease while the CPU's workload increases, as their respective index progresses in the deque. Also, note that while n_p for the CPU (n_p^{CPU}) is fixed, n_p for the GPU (n_p^{GPU}) varies based on the dataset characteristics and on ϵ (Section 4.3.1.2).

As described in Section 4.3, HEGJoin uses two different sorts: sorting by workload (Section 4.3.1.3) and Super-EGO's EGO-sort (Section 4.3.2.2). However, as these two strategies sort following different criteria, it is not possible to first sort by workload then to EGO-sort (and vice versa), as the first sort would be overwritten by the second sort. We thus create a mapping between the EGO-sorted dataset and our shared work queue, as represented in Figure 4.3.

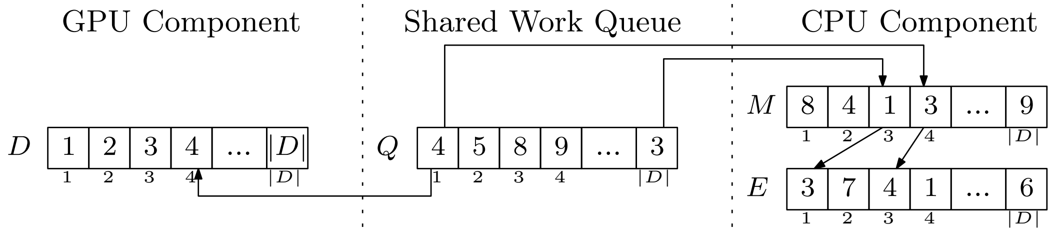


Figure 4.3: Illustration of an input dataset D, the shared deque sorted by workload Q, the input dataset EGO-sorted E and the mapping M between Q and E. The numbers in D, Q, and E correspond to query point ids, while the numbers in M correspond to their position in E. The numbers below the arrays are the indices of the elements.

4.4.2 Workload Partitioning

As previously described, this paper proposes three different methods to partition the work between the CPU and the GPU, using the shared work queue presented in Section 4.4.1. Proposing these three methods allows us to extensively explore work partitioning characteristics, as well as demonstrate the significance of an efficient work partitioning method. Our three partitioning methods use the shared deque presented in Section 4.4.1, as in all three cases the work still needs to be partitioned among threads. Thus, we describe these partitioning strategies as follows.

4.4.2.1 Dynamic Work Partitioning Strategy

Our dynamic work partitioning strategy assigns work to the CPU and GPU on-demand until the queue is empty. Constantly querying the queue for a fraction of work provides good load balancing, as the processors are likely to complete their last batch of queries at roughly the same time. The CPU and GPU are both assigned a batch size large enough to accommodate their relative performance (particularly for the GPU, to achieve good occupancy), as well as to reduce the number of atomic accesses to the queue. However, the batch size for the CPu and GPu is also not too large, so they are not assigned too many query points as it might leave a processor without work to compute while the other one is computing a large batch. Figure 4.2 illustrates how the dynamic partitioning strategy works. We describe the procedure used to assign the query points to the processors as follows:

1. We set the GPU's deque index to 1 and the CPU's deque index to $|D|$.

2. The program terminates if the GPU's and CPU's indices are at the same position in the deque.

3. To assign query points to a GPU stream, we create and assign a new batch of queries to this GPU stream, and increase GPU's deque index.

4. To assign query points to CPU thread, we create and assign a new batch of queries to this CPU thread, and decrease CPU's deque index.

4.4.2.2 Static Partitioning Strategy Based on Query Points

This static work partitioning method splits the number of query points between the processors, using a static partitioning fraction p_q, where $0 \leq p_q \leq 1$. Hence, from p_q and a number of query points ($|D|$), we can determine the number of query points n_q^{GPU} to assign to the GPU and, by extension, to the CPU. This partitioning fraction p_q is determined based on the estimation of the workload w_{est}, as a function of the number of query points and ϵ. We consider for this partitioning strategy the equal workload assumption, that we describe as follows.

Equal Workload Assumption: In this model, we assume that we do not know the workload of each query point. Thus, we consider that each query point has the same workload. For example, if data is largely unstructured, similarly to a uniform distribution, then all query points would have roughly the same amount of work to compute. Then, based on the query throughput of the CPU and GPU, we assign each architecture a fraction of the total number of queries.

Figure 4.4 illustrates the static partitioning strategy based on query points. Using the equal workload assumption, this example shows the case where the model assigns the same number of query points to the CPU and GPU. In this example, we have an input dataset D sorted by workload (Section 4.3.1.3) made of 100 query points ($|D| = 100$) and indexed by

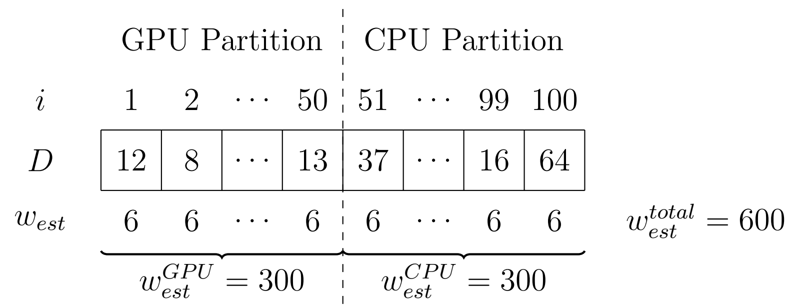


Figure 4.4: Representation of the static partitioning strategy based on the query points, where D is the dataset sorted by workload, i refers to the indices of the query points in D, and W_{est} the workload of the points using the equal workload assumption and that is determined by our model (and where w_{est}^{total} is the estimated workload of HEGJoin). As we use the equal workload assumption, each query point is therefore assigned the same workload. Furthermore, we consider in this example that the model considers the CPU and GPU to have the same throughput, and thus assigns the same estimated workload to both processors ($w_{est}^{CPU} = w_{est}^{GPU}$), i.e., the same number of query points.

i. In this example, our model determines an overall workload $w_{est}^{total} = 600$, which, following the equal workload assumption, corresponds to each point having an estimated workload $w_{est} = w_{est}^{total}/|D| = 6$. We consider in this example that the model determines that the CPU and GPU have the same throughput, and should therefore be assigned the same workload ($w_{est}^{CPU} = w_{est}^{GPU}$), and thus the same number of query points. Depending on the dataset's characteristics (such as its distribution), this estimated workload might differ from the actual workload to compute, which may have an impact on the overall performance of HEGJoin when using such a static partitioning strategy, compared to the other partitioning strategies we propose in this paper.

Using the equal workload assumption, we propose a model to determine the static partitioning fraction p_q for HEGJoin (where $0 \leq p_q \leq 1$). For a specific dataset in d dimensions, we consider a reference search distance ϵ_r, its search volume in d dimensions $v(\epsilon_r) = \frac{\pi^{d/2}}{\Gamma(\frac{d}{2}+1)} \times {\epsilon_r}^{d}$, and the corresponding execution time of LBJoin ($T_{\epsilon_r}^{GPU}$) and Super-EGO ($T_{\epsilon_r}^{CPU}$) when computing the distance similarity join on d with the reference search distance ϵ_r. Hence, for a given search distance ϵ_s for which we want to determine the work partitioning fraction p_q, we predict the execution time of LBJoin and Super-EGO by scaling their execution time $T_{\epsilon_r}^{GPU}$ and $T_{\epsilon_r}^{CPU}$ by the ratio of the search volume $v(\epsilon_s)$ over the reference search volume $v(\epsilon_r)$. The ratio $v(\epsilon_s)/v(\epsilon_r)$ corresponds to the estimated workload increase when the search distance increases as well. Thus, we predict the execution time of the GPU-only algorithm (LBJoin) T^{GPU} as follows:

$$T^{GPU}(\epsilon_s, \epsilon_r, T_{\epsilon_r}^{GPU}) = T_{\epsilon_r}^{GPU} \times \frac{v(\epsilon_s)}{v(\epsilon_r)} \tag{4.1}$$

Similarly, we predict the execution time of the CPU-only algorithm (Super-EGO) T^{CPU} as follows:

$$T^{CPU}(\epsilon_s, \epsilon_r, T_{\epsilon_r}^{CPU}) = T_{\epsilon_r}^{CPU} \times \frac{v(\epsilon_s)}{v(\epsilon_r)} \tag{4.2}$$

We then compute the GPU query throughput (the number of query points the GPU can process per second) as $f_q^{GPU} = |D|/T^{GPU}(\epsilon_s, \epsilon_r, T_{\epsilon_r}^{GPU})$, as well as the CPU query throughput $f_q^{CPU} = |D|/T^{CPU}(\epsilon_s, \epsilon_r, T_{\epsilon_r}^{CPU})$. In addition, we consider the upper bound query throughput as $f_q = f_q^{GPU} + f_q^{CPU}$, and which corresponds to the sum of the GPU and CPU query throughput. Using this upper bound query throughput f_q, we can predict the execution time T^{HEGJoin} of HEGJoin when using any of our static partitioning strategies. We compute this predicted execution time as follows:

$$T^{\text{HEGJoin}} = |D|/f_q \tag{4.3}$$

In addition to predicting the execution time of HEGJoin, we use the upper bound query throughput f_q to determine the static partitioning fraction p_q as the ratio of f_q^{GPU} over f_q. Consequently, we compute the static partitioning fraction as follows:

$$p_q = f_q^{GPU}/f_q \tag{4.4}$$

Finally, we use p_q to determine the number of query points to assign to the GPU as

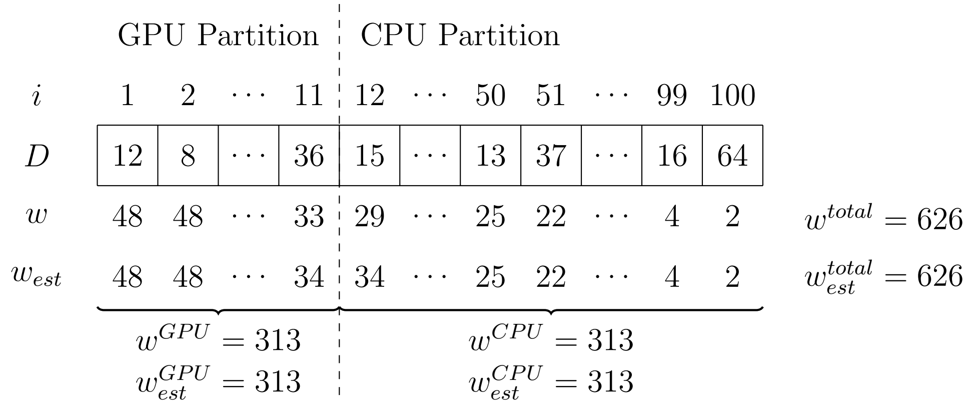
$n_q^{GPU} = |D| \times p_q$. By extension, we determine the number of query points to assign to the CPU as $n_q^{CPU} = |D| - n_q^{GPU}$.

4.4.2.3 Static Partitioning Strategy Based on Candidate Points

Static partitioning based on candidate points considers the total number of candidate points to refine, as well as the number of candidate points of the individual query points. Hence, while the previous static partitioning strategy assumes an equal workload between the query points, we acknowledge here that the query points are likely to each have a different workload. We thus propose the unequal workload assumption as follows.

Unequal Workload Assumption: We consider for this model that each query point can have a workload different from the other query points. Hence, if a dataset has dense regions and sparse regions, the workload that is assigned to the query points is an accurate reflection of their workload in comparison to the *equal workload assumption*.

Figure 4.5 illustrates the static partitioning strategy based on the number of candidate points to refine. In this example, the model considers that the CPU and GPU have the same throughput and should, therefore, be assigned the same number of candidate points to refine. Hence, we have an input dataset D with 100 query points ($|D| = 100$) that are sorted by their respective workload w, and a total number of candidate points to refine $w^{total} = 626$. This model estimates a number of candidate points to refine $w_{est}^{total} = 626$, which is split equally between the CPU and the GPU (as the model considers they have the same throughput in this example). Thus, the GPU is assigned an estimated total number of candidate points to refine $w_{est}^{GPU} = 313$, and $w_{est}^{CPU} = 313$ for the CPU. And, since this model considers the unequal workload assumption, the GPU's workload is the same as its estimated workload ($w^{GPU} = w_{est}^{GPU}$). The same outcome applies to the CPU ($w^{CPU} = w_{est}^{CPU}$). Furthermore, while this example workload corresponds to 11 query points for the GPU and 89 query points for the CPU (determined by the cumulative workload of these query points), the respective total number of candidate points to refine of the GPU and the CPU is similar



	GPU Partition				CPU Partition							
i	1	2	$\cdots$	11	12	$\cdots$	50	51	$\cdots$	99	100	
D	12	8	$\cdots$	36	15	$\cdots$	13	37	$\cdots$	16	64	
w	48	48	$\cdots$	33	29	$\cdots$	25	22	$\cdots$	4	2	$w^{total} = 626$
w_{est}	48	48	$\cdots$	34	34	$\cdots$	25	22	$\cdots$	4	2	$w^{total}_{est} = 626$

$w^{GPU} = 313$ $w^{CPU} = 313$

$w^{GPU}_{est} = 313$ $w^{CPU}_{est} = 313$

Figure 4.5: Representation of the static partitioning strategy based on the candidate points, where D is the dataset, i the indices of the query points in D, the workload of the points w as used to sort them by their workload (and where w^{Total} is the total number of candidate points to refine), and w_{est} the workload of the query points determined by the model (and where w^{total}_{est} is the total estimated workload of HEGJoin). While we consider for this example that the model estimates a workload that is equal to the workload of HEGJoin ($w^{total}_{est} = w^{Total}$), there might be scenarios in which w^{total}_{est} and w^{Total} are different. We consider in this example that the model estimates the CPU and GPU to have the same throughput, and thus assign the same number of estimated candidate points to refine to the CPU and to the GPU ($w^{CPU}_{est} = w^{GPU}_{est}$). Furthermore, as in this example the estimated workload is the same as the actual workload of HEGJoin ($w^{total}_{est} = w^{Total}$), both processors are assigned the same amount of work to compute ($w^{GPU} = w^{CPU}$), i.e., the same number of candidate points to refine.

($w^{GPU} \approx w^{CPU}$). Given that the CPU and GPU are considered to have the same throughput in this example, this strategy should yield a relatively low load imbalance between the CPU and GPU.

This static partitioning strategy uses Equations 4.1 and 4.2 to predict the execution time of LBJoin and Super-EGO for a specific dataset and a given search distance ϵ_s. Hence, we use the total number of candidate points to refine w, as determined when sorting the query points by their workload, in addition to the predicted execution time $T^{GPU}(\epsilon_s, \epsilon_r, T^{GPU}_{\epsilon_r})$ to compute the GPU candidate point throughput (the number of candidate points the GPU can refine per second) $f^{GPU}_c = w/T^{GPU}(\epsilon_s, \epsilon_r, T^{GPU}_{\epsilon_r})$. Similarly to the GPU, we compute the number of candidate points throughput refined by the CPU $f^{CPU}_c = w/T^{CPU}(\epsilon_s, \epsilon_r, T^{CPU}_{\epsilon_r})$. In comparison to the static partitioning based on the query points (Section 4.4.2.2), we compute here the upper bound throughput of the number of candidate points refined $f_c =$

$f_c^{GPU} + f_c^{CPU}$, which corresponds to the sum of the throughput of the number of candidate points refined by the CPU and GPU. We then use this upper bound candidate refinement throughput to determine the static partitioning fraction p_c (where $0 \leq p_c \leq 1$), and which is computed as follows:

$$p_c = f_c^{GPU}/f_c \tag{4.5}$$

We then use this static partitioning fraction p_c to determine the number of candidate points to assign to the GPU, $n_c^{GPU} = w \times p_c$. Similarly, we determine the number of candidate points to assign the CPU, $n_c^{CPU} = w - n_c^{GPU}$. Furthermore, as we consider the unequal workload assumption we described above, we need to find the number of query points to assign to the GPU, n_q^{GPU}, and for which their cumulative workload is the closest to the GPU's assigned workload n_c^{GPU} (by extension, we also find $n_q^{CPU} = |D| - n_q^{GPU}$).

While the GPU and CPU do not use the same indexing method, and thus do not yield the same number of candidate points to refine, our experimental evaluation (Section 4.5) will show that the number of candidate points to refine, w, yielded by the grid indexing schemes in LBJoin (Section 4.3.1) and Super-EGO (Section 4.3.2) are roughly consistent such that we assume w is equal for both indexing schemes.

4.4.2.4 Summary of Work Partitioning Strategies

In this section, we summarize the key points of the work partitioning strategies we presented in sections 4.4.2.1, 4.4.2.2 and 4.4.2.3 above.

- **Dynamic Partitioning Strategy:** This work partitioning strategy uses the shared deque proposed in Section 4.4.1 to assign query points to the CPU and GPU on-demand, until the deque is empty. The main objective of this partitioning method is to have the CPU and GPU finishing their last batch of query points roughly at the same time, particularly by frequently querying the deque for a new batch to compute.

We denote HEGJoin using this dynamic partitioning strategy as HEGJoin-Dyn.

- **Static Partitioning Strategy Based on Query Points:** The static partitioning strategy based on query points that we described in Section 4.4.2.2 estimates the workload of HEGJoin to assign a number of query points to the CPU and GPU. Given a specific dataset, a search distance ϵ_r and the execution time of LBJoin and Super-EGO, this strategy estimates the computation time of HEGJoin by scaling the execution time of LBJoin and Super-EGO using ϵ_r and the search distance used to compute the distance similarity join. From this estimated computation time and the execution time of the GPU-only and CPU-only algorithms, we determine the static partitioning fraction p_q (where $0 \leq p_q \leq 1$), and then the number of query points to assign to the GPU and CPU, assuming all the query points have an equal workload. We denote HEGJoin using this static partitioning strategy based on query points as HEGJoin-SQ.

- **Static Partitioning Strategy Based on Candidate Points:** This static partitioning strategy based on candidate points that we introduced in Section 4.4.2.3 divides the total number of candidate points to refine between the CPU and GPU. Similarly to the partitioning method based on query points, we predict the execution time of LBJoin and Super-EGO the same way as we do for the static partitioning strategy based on query points. However, we use this predicted execution time to determine the static partitioning fraction p_c and splits the total number of candidate points to assign to the CPU and GPU. Hence, we determine the number of candidate points to assign to the GPU and CPU, and then find the number of query points whose cumulative workload is the closest to the workload assigned to the GPU. Similarly, we determine the number of query points to assign to the CPU. We denote HEGJoin using this static partitioning strategy based on candidate points as HEGJoin-SC.

Table 4.1 summarizes the properties of HEGJoin-Dyn, HEGJoin-SQ, and HEGJoin-

30	30	30	30	**20**	22	22	22	**22**	22	22	22
1	2	3	4	5	6	7	8	9	10	11	12

Figure 4.6: Representation of the new batch estimator. The bold numbers are the estimated number of neighbors of those points, while the other numbers are inferred, based on the maximum result between the two closest estimated points shown in bold.

SC. HEGJoin-Dyn and HEGJoin-SC have mutually exclusive properties, whereas HEGJoin-SQ overlaps the properties of HEGJoin-Dyn and HEGJoin-SC. By examining these three work partitioning strategies, we cover a large range of properties, thus enabling us to make a comprehensive examination of work distribution in HEGJoin.

Table 4.1: Summary of the different properties of HEGJoin-Dyn, HEGJoin-SQ, and HEGJoin-SC.

	HEGJoin-Dyn	HEGJoin-SQ	HEGJoin-SC
Workload-oblivious	✓	✓	
Workload-aware			✓
On-demand	✓		
Planned		✓	✓
Architecture-oblivious	✓		
Architecture-aware		✓	✓

4.4.3 Batching Scheme: Complying with Non-Increasing Workload

Because the batching scheme proposed by Gowanlock and Karsin [45] and presented in Section 4.3.1.2 was not designed for non-increasing workloads, we had to adapt it to fit our sort by workload strategy (Section 4.3.1.3) and its non-increasing workload. Indeed, as the batch estimator creates batches with a fixed number of query points, and because the query points are sorted by workload, this batching scheme creates successive batches with a non-increasing workload. Hence, as the execution proceeds, the batches become smaller, take less time to compute, and the overhead of launching many kernels may become substantial, particularly when the computation could have been executed with fewer batches.

We modify the batching scheme (Section 4.3.1.2) to accommodate the sort by workload strategy, and that is represented in Figure 4.6. While still estimating a fraction of the points, the rest of the points get a number of neighbors inferred from the maximum value of the two closest estimated points (to overestimate and avoid buffer overflow during computation). Adding the estimated and the inferred number of neighbors yields an estimated result set size n_e. We then create the batches so they have a consistent result set size r_l close to the buffer size n_s. As the number of estimated neighbors should decrease (as their workload decreases), the number of query points per batch increases.

When using the dynamic partitioning (Section 4.4.2.1), we set a minimum number of batches to $2 \times n_f$, where $n_f = 3$ is the number of CUDA streams used. Therefore, the GPU can only initially be assigned up to half of the queries in the work queue. This ensures that the GPU is not initially assigned too many queries, which would otherwise starve the CPU of work to compute. When using a static partitioning strategy (Sections 4.4.2.2 and 4.4.2.3), then we set the minimum number of batches for the GPU $n_f = 3$, so each CUDA stream has at least a batch to compute. We do not create more batches for the CPU, as it already has its own reserved fraction of the work, determined by one of the static partitioning strategies.

4.4.4 GPU Component: HEGJoin-GPU

The GPU component of our heterogeneous algorithm, which we denote as HEGJoin-GPU and that we can divide into two parts (the host and the kernel), remains mostly unchanged from the algorithm proposed by Gallet and Gowanlock [35] and presented in Section 4.3.

Regarding the host side of our GPU component, we modify how the kernels are instantiated to use the shared work deque presented in Section 4.4.1. Therefore, as the original algorithm was looping over all the batches (as given by the batch estimator, presented in Section 4.3), the algorithm now loops while the shared deque returns a batch to compute (Section 4.4.1).

In the kernel, since the work queue has been relocated to the CPU, a batch corresponds to a range of queries in the deque whose interval is determined when taking a new batch from the queue, and can be viewed as a "local queue" on the GPU. Therefore, the threads in the kernel update a counter that is local to the batch to determine which query point to compute, still following the non-increasing workload that yields a good load balancing between threads in the same warp.

4.4.5 CPU Component: HEGJoin-CPU

The CPU component of HEGJoin, which we denote as HEGJoin-CPU, is based on the Super-EGO algorithm proposed by Kalashnikov [57] and presented in Section 4.3.2. We make several modifications to Super-EGO to incorporate the shared deque we use, and we also optimize Super-EGO to improve its performance. We denote this improved version of Super-EGO as New-Super-EGO.

As described in Section 4.3.2, Super-EGO uses a queue and a producer-consumer system for multithreading. We remove this system and replace it with our shared deque. Because the threads are continuously taking work from the shared deque until it is empty, the producer-consumer system originally used becomes unnecessary, as the deque informs New-Super-EGO when it is empty.

The original Super-EGO algorithm recursively creates sub-partitions of contiguous points on the input datasets until their size is suited for joining. As one of the partitions is now taken from our deque, which is sorted by workload, it no longer corresponds to a contiguous partition of the input dataset. Thus, we loop over the query points of the batch given by the deque to join it with the other points in the partition. This optimization requires the use of the mapping presented in Section 4.4.1 and illustrated in Figure 4.3.

Super-EGO uses qsort from the C standard library to EGO-sort, and we replace it by the more efficient and parallel boost::sort::sample_sort algorithm, a stable sort from the Boost C++ library. This allows New-Super-EGO to start its computation earlier than Super-EGO would, as it is faster than qsort. We use as many threads to sort as we use to compute the join.

Finally, in contrast to the original Super-EGO algorithm, New-Super-EGO is now capable to compute and to store data using 64-bit floats instead of only 32-bit floats.

4.5 Experimental Evaluation

In this section, we present the experimental evaluation we conducted to measure the performance of HEGJoin against the work it leverages (LBJoin and Super-EGO), as well as the efficiency of the different work partitioning strategies we propose in this paper.

4.5.1 Selectivity

We report the selectivity as defined by Kalashnikov [57] of our experiments as a function of ϵ. We define the selectivity $S = (|R| - |D|)/|D|$, where R is the result set and D is the input dataset. The selectivity thus corresponds to the average number of neighbors found per query point, excluding the query points from finding themselves.

4.5.2 Datasets

In this section, we present the real-world and synthetic datasets we use to evaluate the performance of HEGJoin and our partitioning strategies. We detail the real-world datasets that we select as follows:

- *SW-* [69], composed of 1.86M or 5.16M points in two dimensions representing the latitude and longitude of the objects, and adding the total number of electrons as the third dimension.

- *SDSS* [5], composed of a sample of 15.23M galaxies in 2 dimensions.

- *Gaia* [2], in which we select the position of 50M objects from the Gaia catalog.

- *OSM* [1], which is a collection of GPS point data from OpenStreetMap, and in which we also select 50M objects.

Table 4.2: Summary of the datasets used to conduct our experiments. $|D|$ denotes the number of points, d the dimensionality, and S the selectivity range for the values of ϵ we use. The *Expo-* datasets are exponentially distributed synthetic datasets (using $\lambda = 40$), while the others are real-world datasets.

Dataset	$\lvert D\rvert$	d	S	Dataset	$\lvert D\rvert$	d	S
Expo2D2M	2M	2	397–9.39K	*Expo2D10M*	10M	2	80–1.99K
Expo3D2M	2M	3	64–6.70K	*Expo3D10M*	10M	3	9–1.06K
Expo4D2M	2M	4	23–9.26K	*Expo4D10M*	10M	4	3–1.63K
Expo6D2M	2M	6	0–2.68K	*Expo6D10M*	10M	6	0–499
Expo8D2M	2M	8	0–157	*Expo8D10M*	10M	8	0–167
SW2DA	1.86M	2	295–5.82K	*SW2DB*	5.16M	2	91–2.03K
SW3DA	1.86M	3	239–13.20K	*SW3DB*	5.16M	3	33–2.13K
Gaia	50M	2	19–455	*OSM*	50M	2	67–571
SDSS	15.23M	2	1–31				

In addition to the real-world datasets, we conduct experiments on exponentially distributed synthetic datasets made of 2M and 10M points spanning two to eight dimensions (we detail later the datasets used to evaluate the static partitioning in particular). These datasets are named using the dimensions and number of points; for example, *Expo3D2M* is a 3-dimensional dataset containing 2M points. We elect to use an exponential distribution (with $\lambda = 40$) as this distribution contains over-dense and under-dense regions, similarly to the real-world datasets we select. Finally, exponential distributions yield a high load imbalance between the points, and thus should illustrate the performance of HEGJoin when there is a load imbalance between the CPU and GPU. Furthermore, we do not use uniformly distributed datasets, as this type of dataset would not yield load imbalance, as all the query points would have roughly the same workload. We summarize these real-world and exponentially distributed synthetic datasets in Table 4.2.

4.5.3 Methodology

We conduct all our experiments on the two following platforms:

- **Platform 1:** 2 × Intel Xeon E5-2683-v4 (with 2 × 16 cores), 256 GiB of main memory, and an Nvidia Titan X with 12 GiB of global memory.

- **Platform 2:** 2 × Intel Xeon E5-2620-v4 (with 2 × 8 cores), 128 GiB of main memory, and an Nvidia Quadro GP100 with 16 GiB of global memory.

While we systematically present the results of the experiments using Platform 1, we only show the results of the experiments using Platform 2 in Figure 4.13. The code executed by the CPU is written in C++, while the GPU code is written using CUDA. We use the GNU compiler and use the O3 optimization flag for all experiments.

We summarize the different implementations we evaluate as follows. For clarity, we differentiate between similar algorithm components since they may use slightly different experimental configurations. For example, we make the distinction between the CPU component of HEGJoin, HEGJoin-CPU, and the original Super-EGO algorithm.

- LBJoin: the GPU algorithm proposed by Gallet and Gowanlock [35], using 3 GPU streams (managed by 3 CPU threads), 256 threads per block, $n_s = 5 \times 10^7$ key/value pairs, where the dataset is stored as 64-bit floats, and n_p^{GPU} is given by the batch estimator presented in Section 4.4.3. This configuration is used on both platforms.

- Super-EGO: the CPU algorithm developed by Kalashnikov [57], using 32 (16) CPU threads on Platform 1 (Platform 2), and we use 32-bit floats to store the dataset.

- New-Super-EGO: our optimized version of Super-EGO as presented in Section 4.4.5 that uses the sorting by workload strategy, using 32 (16) CPU threads on Platform 1 (Platform 2), and where the dataset is stored as 64-bit floats.

- HEGJoin-GPU: the GPU component of HEGJoin, using the same configuration as LBJoin and one of the work partitioning strategies we propose (Section 4.4.2).

- HEGJoin-CPU: the CPU component of HEGJoin, using the same configuration as New-Super-EGO and one of the work partitioning strategies (with $n_p^{CPU} = 1,024$ when using the dynamic partitioning and the shared deque).

- HEGJoin: the heterogeneous algorithm that combines HEGJoin-CPU and HEGJoin-GPU. HEGJoin-Dyn denotes HEGJoin when using the dynamic partitioning strategy, HEGJoin-SQ denotes HEGJoin when using the static partitioning strategy based on query points, while HEGJoin-SC denotes HEGJoin when using the static partitioning strategy based on candidate points.

HEGJoin-CPU and HEGJoin-GPU, as part of HEGJoin, each compute a fraction of the work. LBJoin, Super-EGO, New-Super-EGO and HEGJoin are standalone algorithms, and thus compute all the work. All response times are averaged over three time trials and include the end-to-end computation time, i.e., the time to construct the grid index on the GPU, sort by workload, reorder the dimensions and to EGO-sort, and the time to join. Note that some of these time components may overlap (e.g., EGO-sort and GPU computation may occur concurrently).

4.5.4 Results

In this section, we present the results of our experimental evaluation. We provide a roadmap for the organization of our results as follows:

- In Section 4.5.4.1, we present the performance of New-Super-EGO when compared to Super-EGO.

- In Section 4.5.4.2, we compare the search space pruning efficiency of the indexes used by New-Super-EGO and LBJoin.

- In Section 4.5.4.3, we outline the accuracy of the models used by HEGJoin-SQ and HEGJoin-SC that we proposed in Sections 4.4.2.2 and 4.4.2.3.

- After the baseline performance has been demonstrated in Sections 4.5.4.1–4.5.4.3, in Section 4.5.4.4 we show the performance of HEGJoin-Dyn, HEGJoin-SQ and HEGJoin-SC, as compared to the leveraged algorithms, New-Super-EGO and LBJoin.

- In Section 4.5.4.5, we evaluate the efficiency of our shared work queue by measuring the load imbalance between the CPU and GPU.

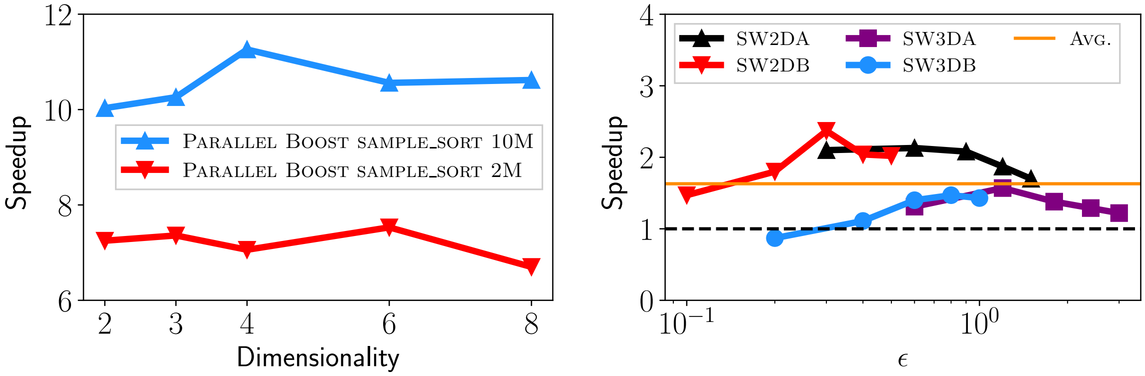
- In Section 4.5.4.6, we assess the overhead incurred by the data transfers between the CPU and GPU when using HEGJoin.

4.5.4.1 Performance of New-Super-EGO

In this section, we evaluate the performance of New-Super-EGO, the optimized version of Super-EGO. The major optimizations include a different sorting algorithm, using the sorting by workload strategy and work queue (Section 4.4.5). The experiments in this section were conducted on a selection of datasets from Table 4.2. The results we show in this section are from using Platform 1.

We evaluate the performance of EGO-sort using the parallel sample_sort algorithm from the C++ Boost library over the qsort algorithm from the C standard library. sample_sort is used by New-Super-EGO (and thus by HEGJoin), while qsort is used by Super-EGO. Figure 4.7(a) plots the speedup of sample_sort over qsort on our synthetic datasets. We observe an average speedup of 7.18× and 10.55× on the 2M and 10M points datasets, respectively. Note that we elect to use the sample_sort as the EGO-sort needs to be stable.

Figure 4.7(b) plots the speedup of New-Super-EGO over Super-EGO on the *SW*-real-world datasets. New-Super-EGO achieves an average speedup of 1.63× over Super-EGO. While New-Super-EGO stores data as 64-bit floats, Super-EGO only uses 32-bit floats and thus has a performance advantage compared to New-Super-EGO. The overall speedup is explained by using sample_sort over qsort, and the sorting by workload strategy with the work queue. Therefore, New-Super-EGO largely benefits from balancing the workload between its threads and from using the work queue.



(a) EGO-Sort Speedup

(b) New-Super-EGO Speedup

Figure 4.7: (a) Speedup to EGO-Sort our exponentially distributed synthetic datasets using sample_sort from the Boost library over qsort from the C standard library. $S = 0$–9.39K and $S = 0$–1.99K on the 2M and 10M points datasets, respectively. (b) Speedup of New-Super-EGO over Super-EGO on the *SW*- real-world datasets. Results from Platform 1.

4.5.4.2 Candidate Point Pruning Efficiency of LBJoin and New-Super-EGO

In this section, we explore the pruning efficiency of the grid when used by LBJoin and when used by New-Super-EGO. As we mentioned in Section 4.4.2.3, because LBJoin and New-Super-EGO use two different grid indexes, the pruning of the search space may yield a different number of candidate points to refine. Hence, we compare in Table 4.3 the number of candidate points refined by LBJoin and New-Super-EGO, as well as the ratio of the number of candidate points refined by LBJoin over the number of candidate points refined by New-Super-EGO on a selection of datasets. We observe that in lower dimensions, the difference in the number of candidate points refined by LBJoin and New-Super-EGO is relatively low, as the ratio is around 1. However, as dimensionality increases, we observe that this ratio tends to decrease, indicating that New-Super-EGO becomes less efficient at pruning the search space than LBJoin. The results we show in this section are from Platform 1.

Table 4.3: Comparison of the number of candidate points refined by LBJoin vs. New-Super-EGO, and ratio of the number of candidate points refined by LBJoin over the number of candidate points refined by New-Super-EGO on a selection of our datasets (Table 4.2). Results from Platform 1.

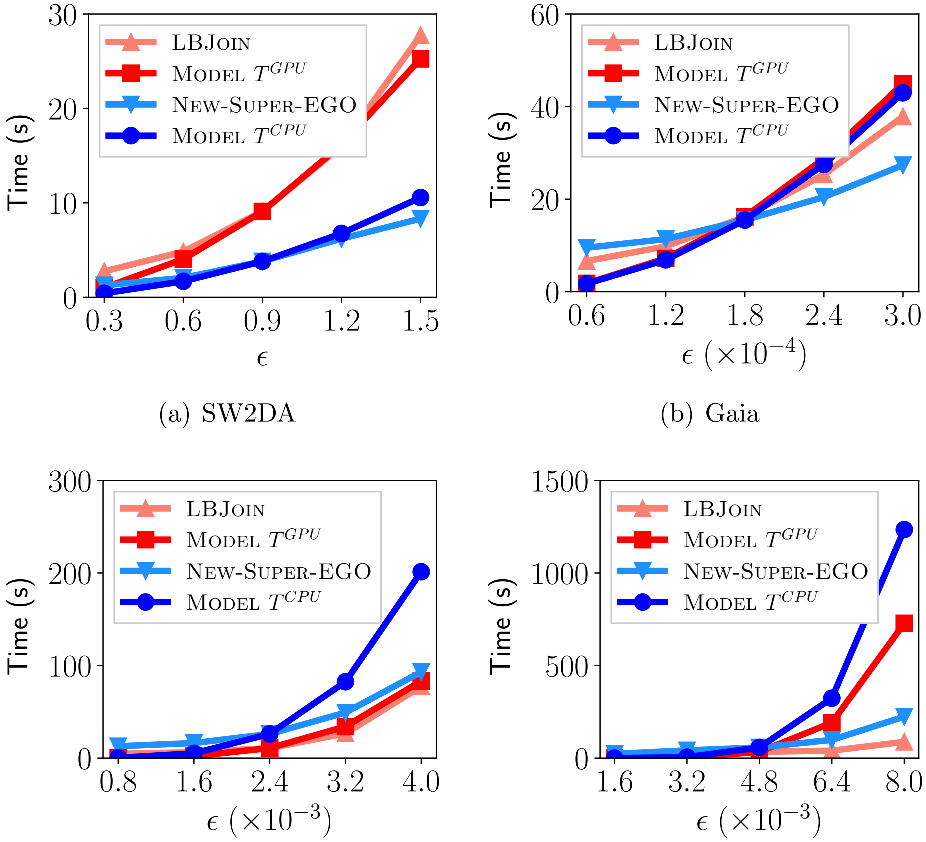
Dataset	ϵ	S	LBJoin	New-Super-EGO	Ratio
SW2DA	1.5	5.82K	28,441,701,752	27,786,778,388	1.02
SDSS	0.002	31	65,531,735,119	66,154,801,616	0.99
SW3DA	3.0	13.20K	90,349,946,258	87,855,196,567	1.03
Expo2D2M	0.002	9.39K	51,789,286,408	50,121,273,123	1.03
Expo2D10M	0.0004	1.99K	56,439,981,246	54,645,837,741	1.03
Expo4D2M	0.01	9.26K	113,929,159,776	177,787,029,288	0.64
Expo4D10M	0.004	1.63K	217,420,698,818	216,050,585,244	1.01
Expo8D2M	0.015	157	77,827,299,052	108,430,322,625	0.72
Expo8D10M	0.012	167	207,650,110,734	374,000,045,202	0.56

4.5.4.3 Model Validation for HEGJoin-SQ and HEGJoin-SC

In this section, we evaluate the accuracy of the models we propose for the static partitioning strategies based on query points (Section 4.4.2.2) and based on the number of candidate points to refine (Section 4.4.2.3). The results we show in this section are from Platform 1.

Figure 4.8 plots the modeled execution time of LBJoin as T^{GPU} and the modeled execution time of New-Super-EGO as T^{CPU} on a selection of datasets. We observe that in 2-D (Figures 4.8(a) and (b)), the model determines an execution time similar to the execution time of LBJoin and New-Super-EGO. In 4-D (Figure 4.8(c)), while the modeled time for LBJoin is accurate, the modeled time of New-Super-EGO is overestimated when $\epsilon > 2.4 \times 10^{-3}$. On the *Expo6D10M* dataset (Figure 4.8(d)), we observe that the modeled time of both LBJoin and New-Super-EGO are overestimated when $\epsilon > 4.8 \times 10^{-3}$. Thus, we observe that the model may sometimes not accurately predict the response time of HEGJoin and, therefore, may yield a poor distribution of the work to the CPU and GPU.

This poor distribution is particularly impactful when the execution time of a processor is overestimated, while the execution time of the other processor is underestimated. As the model yields $f_q = f_c$, the workload of the static partitioning based on the query points is likely to be higher than the workload of the static partitioning based on the number of



(a) SW2DA

(b) Gaia

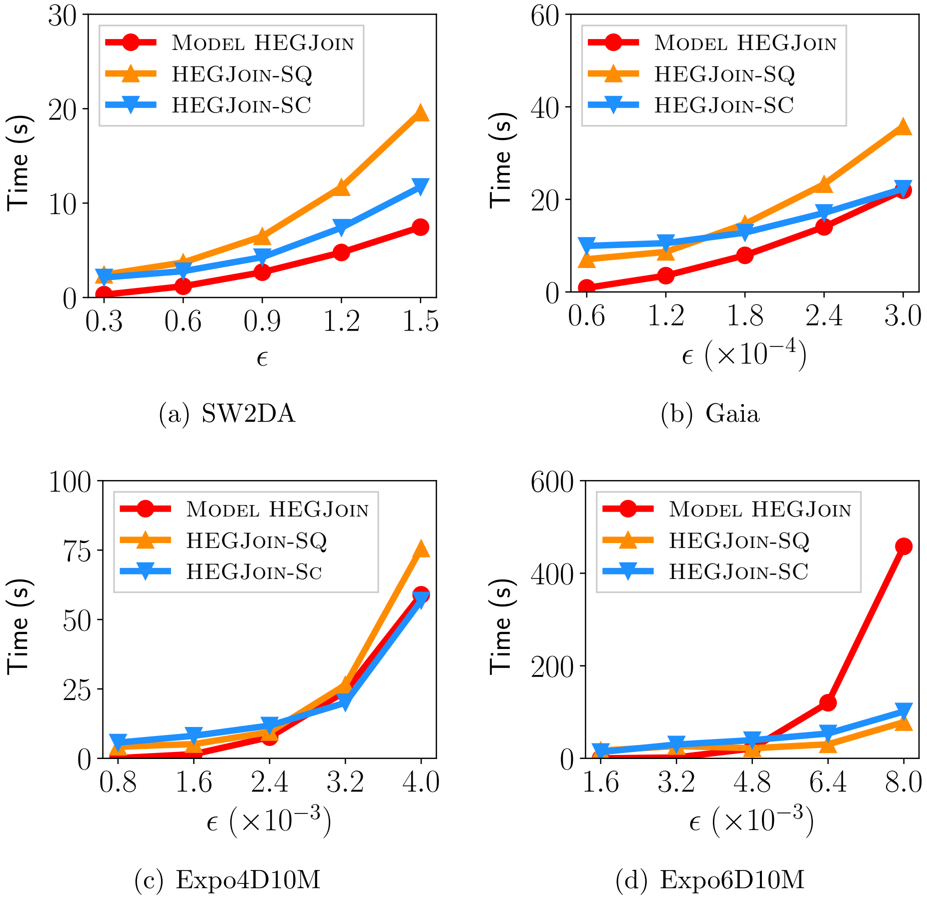
(c) Expo4D10M

(d) Expo6D10M

Figure 4.8: Comparison of the modeled execution times T^{GPU} and T^{CPU} vs. their corresponding reference execution times LBJoin and New-Super-EGO on a selection of datasets: (a) *SW2DA*, (b) *Gaia*, (c) *Expo4D10M* and (d) *Expo6D10M*. Results from Platform 1.

candidate points to refine. Indeed, because the query points are sorted by their workload in a non-increasing order (Section 4.3.1.3), the number of query points determined by the static partitioning fraction f_q will very likely have a cumulative workload higher than the workload yielded by the static partitioning of the candidate points determined by the static partitioning fraction f_c.

Figure 4.9 plots the modeled execution time of HEGJoin as determined by the static partitioning model (Section 4.4.2.2) vs. the response time of HEGJoin-SQ and HEGJoin-SC on a selection of datasets. We observe on the 2-D datasets (Figures 4.9(a) and (b)) that the modeled execution time for HEGJoin is slightly underestimated compared to the exe-



(a) SW2DA

(b) Gaia

(c) Expo4D10M

(d) Expo6D10M

Figure 4.9: Comparison of the modeled execution time of HEGJoin as determined by the static partitioning model vs. the response time of HEGJoin-SQ and HEGJoin-SC on a selection of datasets: (a) *SW2DA*, (b) *Gaia*, (c) *Expo4D10M* and (d) *Expo6D10M*. Results from Platform 1.

cution time of HEGJoin-SQ and HEGJoin-SQ. As we mentioned before, low-dimensional searches are memory-bound, a bottleneck that the model is unable to capture and thus to include in its modeled time. Indeed, as we consider the upper bound throughput as the sum of LBJoin and New-Super-EGO respective throughput, we assume that concurrently using the CPU and the GPU scales perfectly, without said bottlenecks. Nevertheless, the modeled execution time of HEGJoin is overall similar to the execution time of HEGJoin-SQ and HEGJoin-SC. On the *Expo4D10M*, and despite its overestimation of the modeled time of New-Super-EGO (Figure 4.8(c)), the modeled execution time of HEGJoin is very similar to the execution time of HEGJoin-SQ and HEGJoin-SC. Finally, the overestimation of both the modeled time of LBJoin and New-Super-EGO on the *Expo6D10M* dataset (Figure 4.8(d)), is also reflected in Figure 4.9(d), as the modeled execution time of HEGJoin is also overestimated compared to the execution time of HEGJoin-SQ and HEGJoin-SC. However, we observe that the modeled execution time of HEGJoin is very similar to the modeled execution time of LBJoin (Figure 4.8(d)), which means that the model considers, for this dataset, that HEGJoin is mostly relying on the GPU to compute the majority of the work. On the *Expo6D10M* dataset, we would thus expect HEGJoin-SQ and HEGJoin-SC to have a rather high load imbalance, as the CPU is likely to have little work to compute, and therefore to have to wait for the GPU to finish its computation. We confirm this expectation in Section 4.5.4.5 when evaluating the load imbalance of the partitioning strategies we propose.

4.5.4.4 Performance of the Work Partitioning Strategies

In this section, we evaluate the performance of our three work partitioning strategies, i.e., HEGJoin-Dyn, HEGJoin-SQ, and HEGJoin-SC. We compare their performance to LBJoin and New-Super-EGO. While we show results for a selection of our synthetic datasets (Figure 4.10) that span multiple dimensions and size, we show the results on all our real-world datasets (Figure 4.11). The results we show in this section are from Platform 1.

Performance on Exponential Datasets: Figure 4.10 plots the response time of HEGJoin-Dyn, HEGJoin-SQ, HEGJoin-SC, LBJoin and New-Super-EGO on the (a) *Expo2D2M*, (b) *Expo2D10M*, (c) *Expo4D10M*, (d) *Expo6D10M*, (e) *Expo8D2M* and (f) *Expo8D10M* datasets. We select these datasets as they span multiple dimensions and different sizes. We observe on most datasets (Figures 4.10(a)–(d)) that HEGJoin-Dyn and HEGJoin-SC overall yield similar performance, while HEGJoin-SQ is rather inefficient as it does not substantially improve the execution time of either LBJoin or New-Super-EGO. Thus, using the number of candidate points in the model that is used to statically partition the work yields a better work distribution than not considering the candidate points.

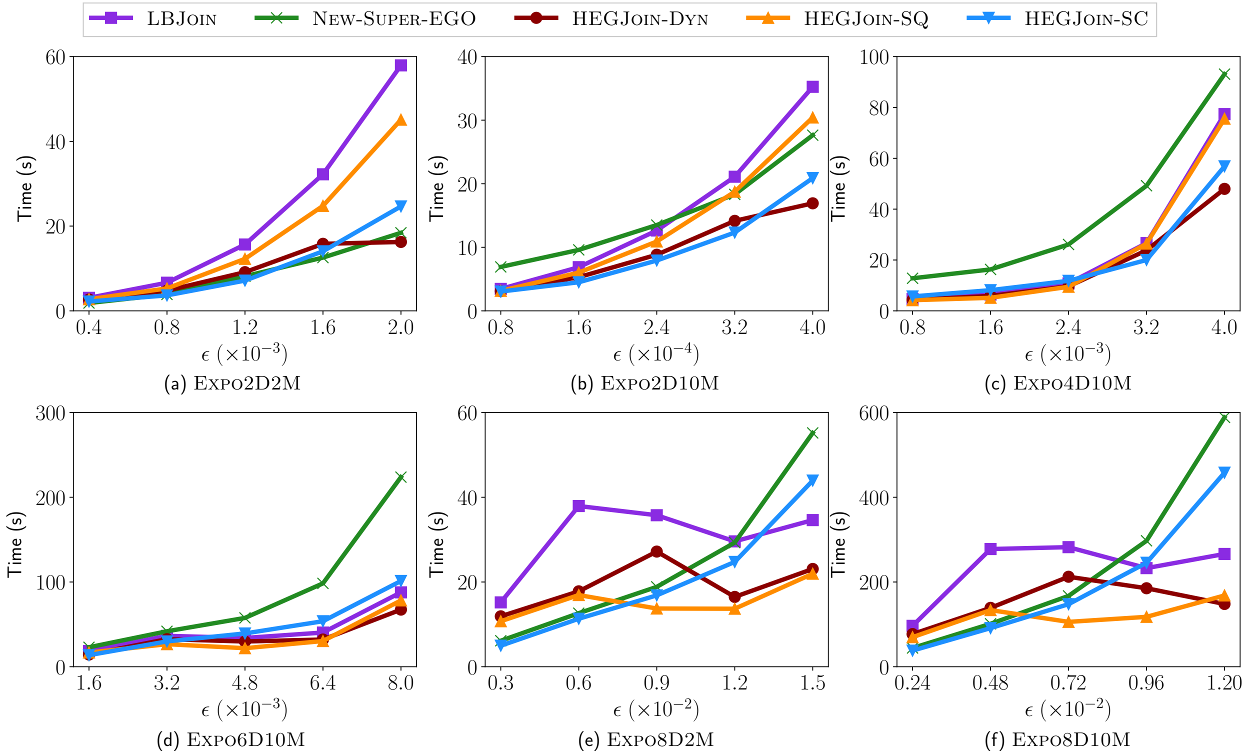


Figure 4.10: Response time of HEGJoin-Dyn, HEGJoin-SQ, HEGJoin-SC, LBJoin and New-Super-EGO on (a) *Expo2D2M*, (b) *Expo2D10M*, (c) *Expo4D10M*, (d) *Expo6D10M*, (e) *Expo8D2M*, and (f) *Expo8D10M*. S is in the range (a) 397–9.39K, (b) 80–1.99K, (c) 3–1.63K, (d) 0–499, (e) 0–157, and (f) 0–167. Results from Platform 1.

In Figure 4.10 we observe on our highest dimensional datasets, and more particularly for the intermediate values of ϵ (*Expo8D2M* where $\epsilon = 0.9 \times 10^{-2}$, and *Expo8D10M* where $\epsilon = 0.72 \times 10^{-2}$), that HEGJoin-Dyn response time does not monotonically increase with ϵ as it is the case in lower dimensions. This occurs because few batches are executed on the GPU, and which take a significant amount of time. This prevents the CPU from taking work from the work queue, thereby increasing the load imbalance between the CPU and GPU. At higher values of ϵ there is less load imbalance between the CPU and GPU; therefore, the response time decreases. On the other hand, we find that on these datasets (*Expo8D2M* and *Expo8D10M*), HEGJoin-SQ is often the most efficient partitioning strategy, while the performance of HEGJoin-SC is between LBJoin and New-Super-EGO. If we examine LBJoin and New-Super-EGO execution times for the median value of ϵ, we observe that the CPU is more efficient than the GPU. Hence, the model assumes that the CPU is consistently more efficient for other values of ϵ, and will therefore assign more work to the CPU. However, the LBJoin execution time does not increase as much as the model predicted, while New-Super-EGO execution time increased more than what the model predicted. Hence, the model will assign a higher fraction of the work to the CPU than it is capable of processing within the execution time estimated by the model. On these particular datasets (*Expo8D2M* and *Expo8D10M*), since the execution time of LBJoin is overestimated and the execution time of New-Super-EGO is underestimated, the static partitioning based on query points ends up being the most efficient partitioning as ϵ increases, since most of the work is assigned to the GPU.

As described in Section 4.1, we choose to focus on low dimensionality. Observe here that the execution time of New-Super-EGO significantly degrades with dimensionality (Figures 4.10(d)–(f)). Therefore, if we were to employ New-Super-EGO at higher dimensions than that explored in this work, the algorithm would have a negligible impact on performance of HEGJoin. In higher dimensions, it would be more worthwhile to consider the use of a different CPU algorithm to replace New-Super-EGO, such as that proposed by Perdacher et al. [82].

Performance on Real-World Datasets: Figure 4.11 plots the response time of HEGJoin-Dyn, HEGJoin-SQ, HEGJoin-SC, LBJoin and New-Super-EGO on the (a) *SW2DA*, (b) *SW3DA*, (c) *SW3DB*, (d) *SDSS*, (e) *Gaia* and (f) *OSM* datasets. We observe on these real-world datasets a similar behavior as on the *Expo2D2M* and *Expo2D10M* datasets (Figures 4.10(a) and (b)). Thus, we observe that the static partitioning based on the number of candidate points to refine, HEGJoin-SC, achieves similar performance as the dynamic partitioning HEGJoin-Dyn. Furthermore, we can see that both of these partitioning strategies achieve similar or better performance than the best performance yielded by LBJoin or New-Super-EGO. Furthermore, we observe that HEGJoin-SQ yields poor performance. As we explained in Section 4.5.4.3, because the execution time may be overestimated or

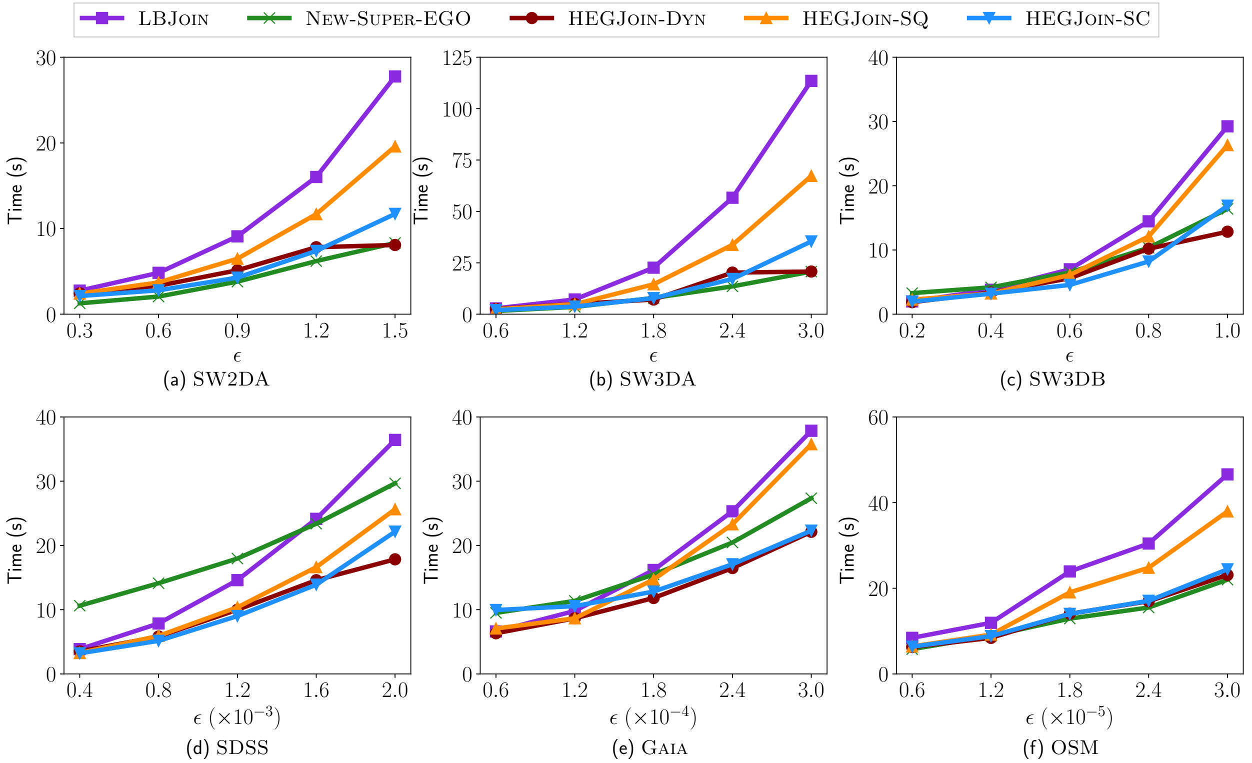


Figure 4.11: Response time of HEGJoin-Dyn, HEGJoin-SQ, HEGJoin-SC, LBJoin and New-Super-EGO on (a) *SW2DA*, (b) *SW3DA*, (c) *SW3DB*, (d) *SDSS*, (e) *Gaia*, and (f) *OSM*. *S* is in the range (a) 295–5.82K, (b) 239–13.2K, (c) 33–2.13K, (d) 1–31, (e) 19–455, and (f) 67–571. Results from Platform 1.

underestimated, a processor can be assigned too much work or too little work relative to its real computational throughput.

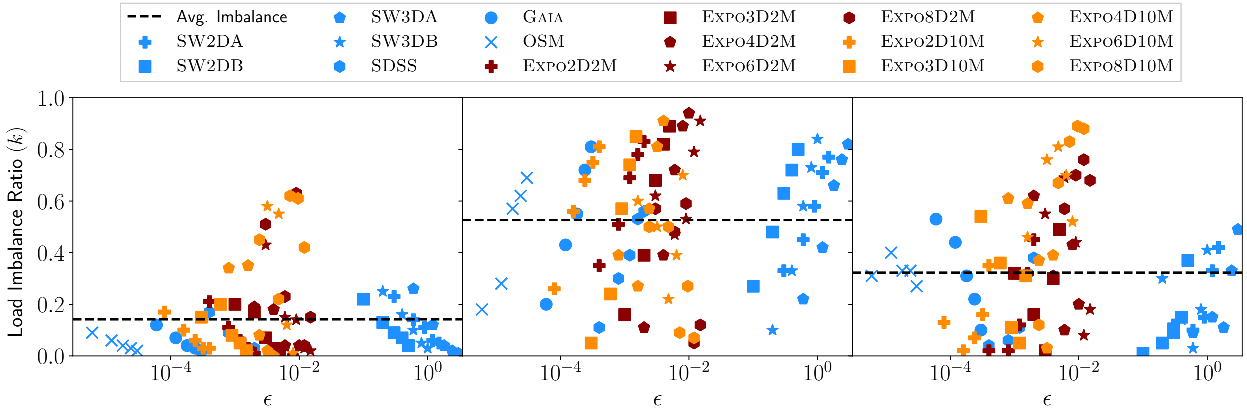
Candidate Point Refinement Throughput: Table 4.4 presents the candidate point refinement throughput (as previously defined in Section 4.4.2.3) for LBJoin, New-Super-EGO, HEGJoin-Dyn, the upper bound (the total throughput given by adding the throughput of the standalone LBJoin and New-Super-EGO algorithms), and the ratio of the throughput HEGJoin-Dyn achieves compared to this upper bound throughput. The candidate throughput corresponds to the number of candidate points to refine, divided by the response time of the algorithm, as shown in Figures 4.10 and 4.11. We observe a relatively high performance ratio, demonstrating that we almost reach the performance upper bound of HEGJoin. Moreover, we also observe that on the *Expo8D10M* dataset, we achieve a

Table 4.4: Throughput of candidate points refined (candidates/s) by LBJoin, New-Super-EGO, the upper bound of LBJoin plus New-Super-EGO, HEGJoin-Dyn, and the performance ratio between HEGJoin-Dyn and the upper bound across several datasets. Results from Platform 1.

Dataset	ϵ	S	LBJoin	New-Super-EGO	Upper Bound	HEGJoin-Dyn	Perf. Ratio
Expo2D2M	0.002	9,392	893,877,929	2,812,071,408	3,705,949,337	3,185,072,965	0.86
Expo4D2M	0.01	9,262	672,847,132	1,777,133,999	2,449,981,131	2,209,642,354	0.90
Expo8D2M	0.015	157	3,881,606,529	1,410,542,112	3,659,149,867	3,372,066,683	0.92
Expo2D10M	0.0004	1,985	1,601,136,521	2,042,081,926	3,643,218,447	3,335,696,291	0.92
Expo4D10M	0.004	1,630	2,809,451,506	2,334,736,697	5,144,188,204	4,531,486,011	0.88
Expo8D10M	0.012	167	2,233,156,849	1,010,004,731	3,243,161,581	4,013,791,675	1.24
SW2DA	1.5	5,818	1,024,556,980	3,419,458,232	4,444,015,212	3,520,012,593	0.79
SDSS	0.002	31	1,798,770,443	2,208,414,897	4,007,185,340	3,673,303,538	0.92
Gaia	0.0003	455	1,696,613,608	2,347,964,353	4,044,577,961	2,903,111,440	0.72
OSM	0.00003	571	1,287,786,499	2,725,941,526	4,013,728,025	2,596,190,956	0.65
SW3DA	3.0	13,207	796,136,506	4,360,015,024	5,156,151,530	4,354,214,277	0.84

ratio of more than 1. We explain this by the fact that *Expo8D10M* is exponentially distributed, and therefore has very dense regions, as well as very sparse regions. Thus, the throughput of LBJoin includes query points with a very low workload, thus increasing its overall throughput compared to what HEGJoin-GPU achieves. Similarly, the throughput of New-Super-EGO includes query points with a very large workload, thus reducing its overall throughput compared to what HEGJoin-CPU achieves. When combining the two algorithms, we have the GPU computing the query points with the largest workload and the CPU the points with the smallest workload. The respective throughput of each component should, therefore, be lower for the GPU and higher for the CPU, than their throughput when computing the entire dataset.

Performance ratios lower than 1 (Table 4.4) indicate that there are several bottlenecks, including contention for memory bandwidth, with the peak bandwidth potentially reached when concurrently storing the results from the CPU and the GPU. We particularly observe this on low dimensionality and for low selectivity, as it yields less computation and higher memory pressure than in higher dimensions or for higher selectivity (Figures 4.10 and 4.11). We confirm this by examining the ratio of kernel execution time over the time to compute all batches of LBJoin. Focusing on the datasets with the minimum and maximum performance



(a) HEGJoin-Dyn (b) HEGJoin-SQ (c) HEGJoin-SC

Figure 4.12: Load imbalance ratio of (a) HEGJoin-Dyn, (b) HEGJoin-SQ and (c) HEGJoin-SC on all the datasets we use for our experiments, and we described in Table 4.2. The horizontal dashed line corresponds to the average load imbalance k, and is as follows: (a) $k = 0.14$, (b) $k = 0.53$ and (c) $k = 0.32$. Results from Platform 1.

ratio from Table 4.4, we find that on *Gaia*, LBJoin has a kernel execution time ratio of 0.06, while on *Expo8D10M*, LBJoin has a kernel execution time ratio of 0.98. Hence, most of the *Gaia* execution time is spent on memory operations, while on *Expo8D10M*, the execution time is mostly spent on computation. When executing LBJoin on *Gaia* (and other datasets with low ratios in Table 4.4), we observe that the use of the GPU hinders the CPU by using a non-negligible fraction of the total available memory bandwidth.

4.5.4.5 Load Balancing Efficiency

We define the load imbalance of HEGJoin as follows. Given the total execution time T, the time t^{GPU} (t^{CPU}) at which the GPU (CPU) ends its work, we characterize the load imbalance ratio as $k = (|t^{GPU} - t^{CPU}|)/T$. A load imbalance ratio close to 0 therefore indicates that the CPU and GPU ended their work at roughly the same time, and thus, that the load imbalance between the CPU and GPU is low. The results we show in this section are from Platform 1.

Figure 4.12 plots the load imbalance ratio of (a) HEGJoin-Dyn, (b) HEGJoin-SQ and (c) HEGJoin-SC across all the datasets we present in Table 4.2. We observe in Figure 4.12(a) that HEGJoin-Dyn (Section 4.4.2.1) achieves a fairly good load balancing, as it achieves an average load imbalance ratio of $k = 0.14$. Furthermore, the datasets in higher dimensions (such as *Expo6D-* and *Expo8D-*) are distinguished by a high load imbalance (with the highest load imbalance ratio, $k = 0.62$, recorded on the *Expo8D10M* dataset and for $\epsilon = 0.72 \times 10^{-2}$). We explain this by the fact that the computation on these datasets is made in only a few large batches (Section 4.3.1.2), and thus explained by the CPU and GPU less frequently accessing the shared deque than in lower dimensions. While having more batches with a reduced size would improve load balancing, it would negatively impact the GPU's performance, as the GPU may be underutilized.

Figure 4.12(b) plots the load imbalance ratio of the static partitioning based on query points, HEGJoin-SQ (Section 4.4.2.2). We immediately observe a high average load imbalance of $k = 0.53$, meaning that on average, the CPU or GPU spend half of the execution time idle. Hence, HEGJoin-SQ yields a load imbalance of up to $k = 0.91$ on the *Expo4D10M* dataset when $\epsilon = 4.0 \times 10^{-3}$. Considering that HEGJoin-SC is usually more efficient than HEGJoin-SQ (Figures 4.10 and 4.11) and yet uses the same model to predict the execution time, the high load imbalance of HEGJoin-SQ is therefore explained by how the work is partitioned between the CPU and GPU, based on query points. Indeed, as we explained above (Section 4.5.4.4), on datasets such as *Expo8D2M* and *Expo8D10M* (Figures 4.10(e) and (f)), as the query points are sorted by workload, the workload assigned to the GPU when partitioning based on query points is higher than the workload assigned when partitioning based on the number of candidate points to refine. Examining the average load imbalance across all values of ϵ of the *SW2DA* dataset, we observe that the average load imbalance is $k = 0.12$ for HEGJoin-Dyn, $k = 0.22$ for HEGJoin-SC, and $k = 0.57$ for HEGJoin-SQ. However, on the *Expo8D10M* dataset and for all the values of ϵ we experiment on this dataset with, the average load imbalance is $k = 0.46$ for HEGJoin-Dyn, $k = 0.66$ for HEGJoin-SC, while $k = 0.29$ for HEGJoin-SQ. Hence, the situations where HEGJoin-SQ achieves a low load imbalance are essentially exceptions to the rather bad performance of HEGJoin-SQ, as they are the result of the model's inaccuracy in such situations.

Figure 4.12(c) plots the load imbalance ratio of HEGJoin-SC, i.e., HEGJoin using the static partitioning based on the number of candidate points to refine (Section 4.4.2.3). We observe an average load imbalance ratio of $k = 0.32$, which is between the average load imbalance ratio of HEGJoin ($k = 0.14$) and HEGJoin-SQ ($k = 0.53$). Furthermore, the highest load imbalance yielded by this static partitioning strategy is on the *Expo8D10M* dataset when $\epsilon = 0.96 \times 10^{-2}$, where $k = 0.89$. The issue on this dataset is the same as when partitioning based on query points: the model is not able to predict situations where the execution time of one of the processors does not increase as the model predicts it will (in this case the execution time of LBJoin). Considering that the execution time of LBJoin is overestimated and that the execution time of New-Super-EGO is underestimated, the GPU is assigned a lower workload and the CPU a higher workload than what they are able to process within the modeled execution time. Finally, and despite HEGJoin-SC having a higher load imbalance than HEGJoin-Dyn, we observe that HEGJoin-SC is roughly as efficient as HEGJoin-Dyn on many datasets and values of ϵ.

4.5.4.6 Data Transfer Overhead of HEGJoin

In this section, we evaluate the overhead of the data transfers between the CPU's main memory and the GPU's global memory, regardless of their direction (from the CPU to the GPU, and vice versa), which is known to be a bottleneck due to the relatively low memory bandwidth of the PCIe-3 interconnect [62]. In Table 4.5 we report the time taken by all the data transfers between the CPU and GPU, the total execution time, and the ratio of the data transfers time to the total execution time, across a selection of datasets and ϵ values. The values are recorded when using HEGJoin-Dyn on Platform 2, over a single trial, and are measured using the Nvidia Visual Profiler. A ratio close to zero indicates that the data transfers are negligible relative to the total execution time of the algorithm, while higher ratios account for a large fraction of the total execution time and thus may degrade

Table 4.5: Total time taken by data transfers between the CPU and the GPU, the total execution time of the algorithm, and the upper bound overhead ratio of the data transfers time to the execution time when using HEGJoin-Dyn. A ratio close to zero indicates an insignificant overhead incurred by data transfers compared to the total execution time. The ratios do not account for the periods of time where the data transfers and the kernel executions overlap. The times were recorded on the Nvidia Visual Profiler over a single time trial using Platform 2.

Dataset	ϵ	S	Data transfer time (s)	Execution time (s)	Upper bound overhead ratio
Expo2D2M	0.002	9,342	5.11	30.99	0.16
Expo4D2M	0.01	9,262	9.22	81.97	0.11
Expo8D2M	0.015	157	0.22	18.47	0.01
Expo2D10M	0.0004	1,985	6.69	27.16	0.25
Expo4D10M	0.004	1,630	8.21	55.27	0.15
Expo8D10M	0.012	167	1.19	116.61	0.01
SW2DA	1.5	5,818	2.66	13.91	0.19
SDSS	0.002	31	7.72	29.74	0.26
Gaia	0.0003	455	9.77	37.83	0.26
OSM	0.00003	571	8.73	36.98	0.24
SW3DA	3.0	13,207	9.41	40.73	0.23

performance. Recall that we use three streams to overlap data transfers with computation (Section 4.3.1.2). However, since there is not a direct way to account for the overlap of data transfers with kernel execution (computation), we consider here that no such overlap occurs. Therefore, ratios we report in Table 4.5 capture the upper bound (worst-case) data transfer overhead.

We observe in Table 4.5 that the ratios of the data transfers time to the total execution time of HEGJoin-Dyn are relatively low across our experiments, despite only capturing the upper bound as described above. Furthermore, the experiments with the highest selectivity (e.g., *SW3DA* for $\epsilon = 3.0$) or the largest datasets (e.g., *Gaia*) yield the highest overhead ratios, due to the large result set size that must be transferred from the GPU to the CPU, or large datasets that must be transferred from the CPU to the GPU. However, in these cases, the relatively high selectivity also yields a large number of batches to compute, making it easier to overlap data transfers with kernel executions, which we could not account for here. On experiments with a lower selectivity (e.g., *Expo8D10M* when $\epsilon = 0.012$), the data transfers account for an insignificant amount of time compared to the high total execution time of HEGJoin-Dyn, due to the relatively small size of the result set that needs to be transferred from the GPU to the CPU. Overall, and given that the overhead ratios in Table 4.5 consist of an upper bound, we consider that the data transfers between the CPU and the GPU are marginally impacting the performance of HEGJoin.

4.5.5 Discussion

We summarize and discuss the major research findings in this paper. We find that HEGJoin using the on-demand work queue (HEGJoin-Dyn) outperforms the two static partitioning methods (HEGJoin-SQ and HEGJoin-SC) on most values of ϵ. Despite this finding, HEGJoin-Dyn does not achieve low load imbalance between the CPU and GPU components of the algorithm across all experimental scenarios (e.g., in Figure 4.12 there is a mean load imbalance of $k = 0.14$). Therefore, dynamically assigning work to the CPU and GPU components of the algorithm is challenging, even when distributing the work on-demand. Part of the reason load imbalance occurs is because the performance characteristics are fundamentally data-dependent regardless of the self-join algorithm (assuming such an algorithm uses the search-and-refine strategy). Query points have differing amounts of work to compute, so it is difficult to split the work and obtain low load imbalance between the CPU and GPU regardless of the method used to distribute the work.

Regarding our performance models, we find that individually modeling the response time of New-Super-EGO and LBJoin is accurate in some cases, and inaccurate in others (Figure 4.8). We constrained the model to only require a single time measurement of New-Super-EGO and LBJoin on each dataset. This restriction means that if the response time increases non-linearly as a function of the search volume, then the model is unable to adequately capture the measured response time. This led to the models overestimating the response time in some cases, yielding a poor distribution of work between the CPU and GPU for the static splitting strategies (HEGJoin-SQ and HEGJoin-SC).

Our static partitioning strategies that distribute the work based on the performance models considered: (*i*) all query points have an identical amount of work to compute (HEGJoin-SQ); and, (*ii*) query points have a varying amount of work to compute based on the size of each query point's candidate set (HEGJoin-SC). A good distribution of work to the CPU and GPU requires that the models are able to adequately capture performance, and we demonstrated this by showing that the partitioning strategy based on (*ii*) outperforms (*i*) above. Despite HEGJoin-SC being able to capture the number of candidate points that need to be refined per query point, we find that the model was unable to capture several performance characteristics that degraded the performance of this static partitioning strategy in some cases. We outline some factors that contribute to poor model accuracy as follows.

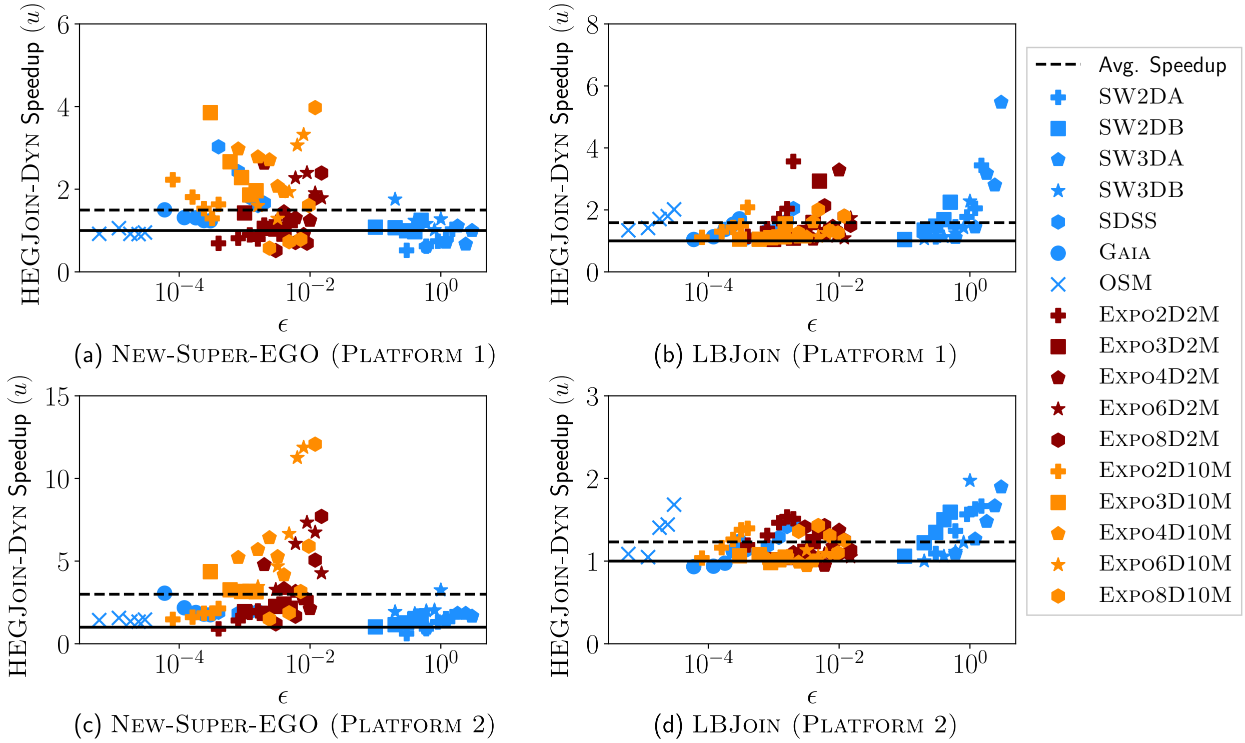
1. The size of the GPU batches must be substantially larger to saturate GPU resources; therefore, this increases the chances that the CPU will be starved of work towards the end of the computation, leading to non-negligible load imbalance.

2. The GPU component of HEGJoin reduces the main memory bandwidth of the CPU component; therefore, if ϵ and data properties lead to a memory-bound execution, the GPU's memory operations will reduce the CPU's available memory bandwidth, which will lead to load imbalance.

3. Depending on data properties and ϵ, the GPU may be underutilized due to many factors, including those related to the SIMT architecture. Typically this occurs when we observe that the response time is roughly "flat" with increasing ϵ (Figure 4.10(e) and (f)). Since the GPU may be underutilized, increasing ϵ has little impact on performance, which causes the model to overestimate the response time. An example of this is shown in Figure 4.9(d) by comparing the execution time of HEGJoin-SC and model curves.

4. Algorithms for the CPU typically achieve the best performance (lowest response time) if they are work-efficient. However, the GPU's architecture can break this work-efficient

assumption, as algorithms designed for the GPU may be work-inefficient but achieve a lower response time than a work-efficient algorithm that performs the same task [27]. Consequently, modeling the performance of a GPU-only algorithm is challenging, and the addition of a concurrently executing CPU algorithm exacerbates this problem.



(a) New-Super-EGO (Platform 1)

(b) LBJoin (Platform 1)

(c) New-Super-EGO (Platform 2)

(d) LBJoin (Platform 2)

Figure 4.13: Average speedup of HEGJoin-Dyn over (a) New-Super-EGO and (b) LBJoin on Platform 1, and over (c) New-Super-EGO and (d) LBJoin on Platform 2, across all datasets (Table 4.2). The horizontal dashed line corresponds to the average speedup u, and is as follows: (a) $u = 1.50$, (b) $u = 1.59$, (c) $u = 2.99$, and (d) $u = 1.23$. The horizontal solid line corresponds to a speedup of $u = 1.0$.

In summary, this paper yields insight into the self-join as executed on heterogeneous architectures, which necessitates a comprehensive examination of the problem of work distribution between architectures. The insights described above outline several challenges related to splitting the work using static and dynamic partitioning strategies. Despite these aforementioned challenges, HEGJoin is more robust to dataset characteristics and search distance, as we find that the algorithm generally outperforms the CPU/GPU-only counterparts. We show the speedup of HEGJoin over New-Super-EGO and LBJoin across all our datasets (Table 4.2), and evaluated on two different platforms in Figure 4.13. We thus observe that HEGJoin-Dyn is, independently from the platform we used, on average more efficient than LBJoin and New-Super-EGO.

4.6 Conclusion

In this paper, we propose HEGJoin, which is to the best of our knowledge the first data-parallel heterogeneous and concurrent CPU-GPU algorithm that computes distance similarity searches, and that leverages LBJoin and Super-EGO, two state-of-the-art algorithms to compute distance similarity searches on the GPU and CPU, respectively. While the computation of distance similarity searches is memory-bound in lower dimensions, it becomes compute-bound in higher dimensions. In both of these situations, the GPU is very suitable at computing distance similarity searches, due to its higher computational throughput and memory bandwidth compared to the CPU.

We propose three work partitioning strategies to assign work to the CPU and GPU; particularly, we propose a dynamic work partitioning strategy that assigns work to the CPU and GPU on-demand through a shared deque, in addition to two static partitioning strategies based on the number of query points, and based on the number of candidate points that will need to be refined. The dynamic partitioning strategy simply does not consider the overall workload of HEGJoin, and is efficient because of the shared deque and its on-demand work assignment to the CPU and GPU. In contrast, the static partitioning strategy HEGJoin-SQ is workload-oblivious, while HEGJoin-SC is workload-aware.

We described several insights into the work partitioning problem between the CPU and GPU based on the static partitioning strategies. To summarize, the use of two different architectures, combined with two different algorithms makes modeling HEGJoin a challenging task. This led to dynamic partitioning being generally more efficient than the two static partitioning strategies. Despite the challenges of statically partitioning the work, we find that HEGJoin with the dynamic deque is more robust to data distributions and the search radius of the self-join than the CPU-only and GPU-only algorithms. Consequently, HEGJoin outperforms the CPU/GPU-only algorithms in most experimental scenarios.

The dynamic partitioning strategy achieved the best performance. Future work should examine different ways to enhance the dynamic partitioning method described in this paper, while still being able to accommodate the GPU's requirement of processing large batches of work to achieve high search throughput. By narrowing our focus on this task, we may be able to further reduce the load imbalance observed, particularly on higher dimensional datasets. Another research direction is to use non-parametric models to statically split the work between the CPU and GPU, which may be able to better capture the complexity of the algorithm. Another possibility is to use an adaptive model that could systematically select the best work partitioning strategy based on data characteristics.

Acknowledgements

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

Chapter 5

Leveraging GPU Tensor Cores for Double Precision Euclidean Distance Calculations

This chapter consists of the peer-reviewed article appearing in the Proceedings of the 29th IEEE International Conference on High Performance Computing, Data, and Analytics (HiPC) [38].

Abstract

Tensor cores (TCs) are a type of Application-Specific Integrated Circuit (ASIC) and are a recent addition to Graphics Processing Unit (GPU) architectures. As such, TCs are purposefully designed to greatly improve the performance of Matrix Multiply-Accumulate (MMA) operations. While TCs are heavily studied for machine learning and closely related fields, where their high efficiency is undeniable, MMA operations are not unique to these fields. More generally, any computation that can be expressed as MMA operations can leverage TCs, and potentially benefit from their higher computational throughput compared to other general-purpose cores, such as CUDA cores on Nvidia GPUs. In this paper, we propose the first double precision (FP64) Euclidean distance calculation algorithm, which is expressed as MMA operations to leverage TCs on Nvidia GPUs, rather than the more commonly used CUDA cores. To show that the Euclidean distance can be accelerated in a real-world application, we evaluate our proposed TC algorithm on the distance similarity

self-join problem, as the most computationally intensive part of the algorithm consists of computing distances in a multi-dimensional space. We find that the performance gain from using the tensor core algorithm over the CUDA core algorithm depends weakly on the dataset size and distribution, but is strongly dependent on data dimensionality. Overall, TCs are a compelling alternative to CUDA cores, particularly when the data dimensionality is low (≤ 4), as we achieve an average speedup of 1.28× and up to 2.23× against a state-of-the-art GPU distance similarity self-join algorithm. Furthermore, because this paper is among the first to explore the use of TCs for FP64 general-purpose computation, future research is promising.

5.1 Introduction

Tensor cores (TCs) are a type of Application-Specific Integrated Circuit (ASIC), and are specifically designed for Matrix Multiply-Accumulate (MMA) operations. The high specificity of TCs makes them typically more efficient at computing MMA operations, than other more general-purpose cores such as CPU cores or GPU CUDA cores. Given four matrices A, B, C, and D, TCs are designed to compute $D = A \times B + C$ (where C and D may be the same matrix). Over the past few years, TCs have been heavily used for machine learning and other fields requiring linear algebra, and few papers have examined broadening the use of TCs for other algorithms. Despite their high specificity, TCs may also be very versatile: any computation expressed with MMA operations, as defined above, should be able to leverage TCs and consequently, benefit from their high computational throughput.

Several companies have proposed a version of TCs, each with its own different characteristics. In this paper, we focus on the Nvidia GPU TCs. These TCs were first introduced with the Volta generation in 2017[1]. Since this first iteration, they have been implemented in several GPU models and have greatly improved over time [72, 73, 76]. In particular,

[1] https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf

while the first generation of TCs was only capable of computing in half precision using 16-bit floats (FP16), TCs are now capable of double precision computing using 64-bit floats (FP64) with the Ampere generation [73]. This enables TCs to be used for applications where high precision is critical. Furthermore, their number, as well as their theoretical computational throughput, have continued to increase, making them an attractive alternative to the general-purpose CUDA cores.

As mentioned above, in this paper we focus on TCs proposed by Nvidia on their GPUs. In addition to the CUDA API to access GPU functionalities, we also leverage the Warp Matrix Multiply-Accumulate (WMMA) API [8, 78], which provides programmatic access to TCs. While other libraries also give access to TCs, they are all higher level than the WMMA API and less versatile, thus less suited to our use case. However, there are some limitations when using the WMMA API. In particular, matrix sizes are limited to a few options, and not all compute precisions are available or can be combined (e.g., FP32 for both multiplication and accumulation is not available, and FP16 multiplication can not be combined with FP64 accumulation).

The Euclidean distance is a metric commonly used in many scientific applications, particularly for data analysis algorithms such as the distance similarity self-join [37, 44, 57, 82], the kNN [41], or the Density-Based Spatial Clustering of Applications with Noise (DBSCAN) [32] algorithms. Within these algorithms, distance calculations are usually the most time-consuming fraction of the total computation [85]. In this paper, we propose to improve the throughput of Euclidean distance calculations by leveraging TCs on the GPU, and consequently also improve the overall performance of the algorithms mentioned above. To illustrate greater applicability to these other algorithms, we use the distance similarity self-join algorithm as a representative example case for the other data analysis algorithms mentioned above. Given a dataset V in d dimensions, the distance similarity self-join algorithm finds all pairs of points (a, b) that are within a distance threshold ϵ of each other; $dist(a, b) \leq \epsilon$, where $a, b \in V$, and $dist$ is the Euclidean distance function.

This paper makes the following contributions:

- We propose a new algorithm for computing Euclidean distances using TCs, leveraging the Nvidia Ampere architecture TCs [73] supporting double precision (FP64) computations.

- We integrate the aforementioned method into the distance similarity self-join algorithm, that we name Tensor Euclidean Distance Join (TED-Join). We show that TED-Join is competitive with the best parallel distance similarity self-joins in the literature for multi-core CPUs and GPU CUDA cores.

- The solution we propose here extends beyond the distance similarity self-join algorithm and can be integrated into other algorithms that use the distance similarity self-join, or more generally Euclidean distance calculations, as a building block.

- We evaluate TED-Join across a broad range of datasets, that span several distributions, sizes and dimensionalities, and compare it to a state-of-the-art GPU CUDA cores (GDS-Join [44]) and two multi-core CPU distance similarity join algorithms, Super-EGO [57] and FGF-Hilbert [82]. We conclude that TED-Join should always be preferred over Super-EGO and FGF-Hilbert, and should be preferred over GDS-Join when the dimensionality $d \leq 4$, where it achieves an average speedup of 1.28× (and 1.07× when considering all the experiments we conducted), and up to 2.23×.

- To our knowledge, this paper proposes the first Euclidean distance calculation for TCs using FP64 computation, and the first use of TCs for the distance similarity self-join.

The paper is outlined as follows: we present essential material in Section 5.2, including an overview of TCs. We then present in Section 5.3 our solution that uses TCs to compute Euclidean distances and its integration into the distance similarity self-join algorithm. We show in Section 5.4 the performance of our solution compared to the state-of-the-art distance similarity self-join algorithms, and we conclude and propose future research directions in Section 5.5.

5.2 Background

5.2.1 Problem Statement

For two points a and b in d dimensions, and where a_i represents the i^{th} coordinate of the point a, and where $i = 1, \dots, d$, the Euclidean distance between a and b is defined as follows:

$$dist(a,b) = \sqrt{\sum_{i=1}^{d}(a_i - b_i)^2}. \tag{5.1}$$

The distance similarity self-join algorithm, as described above, takes a dataset V in d dimensions as well as a search distance ϵ as inputs, and finds all the pairs of points (a, b) such that $dist(a, b) \leq \epsilon$ where $a, b \in V$, and where the distance function is, in this case, the Euclidean distance defined in Equation 5.1. For a query point a, finding all the other points in V within ϵ from a is called a range query ($|V|$ range queries in total).

5.2.2 Tensor Cores (TCs)

TCs on GPUs are an Application-Specific Integrated Circuit (ASIC) designed for Matrix Multiply-Accumulate (MMA) operations. Given four matrices A, B, C and D, this MMA operation is expressed as $D = A \times B + C$. Matrices C and D are the accumulators and may be equivalent. In hardware, TCs are designed to process 4×4 MMA operations. However, the WMMA API only gives access to larger matrices (e.g., 16×16). Therefore, several TCs are used concurrently to perform MMA operations larger than 4×4. Due to their highly specific design, TCs are significantly more efficient at MMA operations than CUDA cores: double precision computation is presented as twice as efficient when using TCs compared to CUDA cores on the Nvidia A100 GPU [73]. This significantly higher processing throughput is our motivation to transform Euclidean distance calculation into MMA operations, and yield higher computational throughput.

The WMMA API [8, 78] provides some low-level access to TCs, giving us the highest versatility possible. However, several limitations come along with this WMMA API. In particular, it is limited to certain matrix sizes and compute precisions. Among the options available, only a few are relevant to our work. In this paper, we focus on FP64 computation, which limits us to only one size for each of our matrices. Let $M_{m,n}$ be a matrix with m rows and n columns. The matrices that we can use with double precision are thus $A_{8,4}$, $B_{4,8}$, $C_{8,8}$, and $D_{8,8}$. We refer the reader to the documentation [78] for the other TCs options.

Programmatically, the matrices proposed by the WMMA API are called *fragments*, and are stored into the GPU threads registers. The WMMA API defines several functions:

- *load_matrix_sync()*: Load a matrix fragment from memory.
- *store_matrix_sync()*: Store a matrix fragment into memory.
- *mma_sync()*: Perform an MMA operation using TCs.
- *fill_fragment()*: Fill a matrix fragment with a specified value.

As their name suggests, these function calls are synchronized. Hence, all 32 threads of the warp are blocked until the operation is complete. The *load* and *store* functions take, among other arguments, a stride between the elements comprising the matrix rows. Hence, all the elements consisting of a row in the target matrix need to be coalesced in memory. Furthermore, the individual elements of the matrix fragments are stored in an unspecified order in the registers. Thus, contrary to regular arrays, the first element of a matrix may not be stored in the first element of the fragment. Consequently, operations on an individual element of a matrix fragment need to be applied to all the other elements, using a loop iterating over all of the elements of the fragment.

5.2.3 Tensor Cores in the Literature

As mentioned above, the literature concerning TCs heavily revolves around machine learning and other closely related fields, and not many other types of applications employing TCs [4, 27, 56, 63, 67]. We present in this section a selection of papers that discuss the use of TCs for applications that focus more on computational/data-enabled science, similarly to this paper. Moreover, since most of the literature seems to focus on low precision computations, we believe that this paper is the first to propose an implementation using TCs for FP64 computations.

Dakkak et al. [27] propose a method to perform reduction and scan operations, using the WMMA API to leverage TCs. Their reduction algorithm consists of multiplying a matrix whose first row are ones and the rest are zeros with a matrix containing the values to reduce, and accumulated with a matrix containing the result from previous reductions. Their scan solution is similar but uses an upper triangular matrix filled with ones and where the rest are zeros, instead of a single row filled with ones. Their proposed solutions achieve a speedup of 100× for the reduction and 3× for the scan, compared to other state-of-the-art methods not using TCs.

Ji and Wang [56] propose using TCs to improve the performance of the DBSCAN algorithm. They mainly use TCs to compute distance matrices between the points that might form a cluster, using the cosine similarity formula (in contrast to the Euclidean distance used in this paper). They also use TCs to perform reductions, which are used to determine if points belong to a cluster or not. Their solution using TCs achieves a speedup of up to 2.61× to compute distance matrices compared to using the CUDA cores. While this work is very relevant to us, it differs in that they use a different distance metric (cosine similarity vs. Euclidean distance), they do not use an index structure, and part of their work is exclusive to the DBSCAN algorithm. In comparison, our solution essentially concerns the Euclidean distance calculations and, therefore, more applications than the distance similarity self-join that we just take as an example for this paper.

Ahle and Silvestri [4] theorize using TCs to compute similarity searches. They use TCs to compute either the Hamming, squared L_2 distances, or cosine similarity through an inner product operation, expressed as matrix multiplications. Additionally, they opt for the Local Sensitivity Hashing (LSH) method, reducing the overall complexity of the computation similarly to an indexing structure used by other similarity join solutions [44, 57, 82]. However, and contrary to these solutions, the LSH method typically yields an approximate result.

5.2.4 Distance Similarity Joins

We discuss in this section several state-of-the-art parallel distance similarity self-join algorithms [37, 44, 57, 82], which we use as reference implementations for our experimental evaluation. These selected algorithms have in common that they use an indexing structure to prune the number of distance calculations, which is a commonly used optimization [21, 22]. When using an index, it is first *searched* to yield a set of candidate points for each query point. The set of candidate points is then *refined* using distance calculations to keep pairs of query and candidate points that are within ϵ of each other.

Kalashnikov [57] proposes Super-EGO, a parallel CPU algorithm to compute a distance similarity join, which is an improvement over the Epsilon Grid Order (EGO) algorithm proposed by Böhm et al. [21]. Super-EGO performance relies on a grid index and which is dependent on the search distance ϵ, where a grid with cells of size $\epsilon \times \epsilon$ is laid on the search space to efficiently prune the candidate points to refine. Furthermore, the author proposes to reorder the dimensions of the points based on their variance, so dimensions with the highest variance are considered first when computing the distance between two points. Hence, their cumulative distance is more likely to reach ϵ sooner, allowing the short-circuiting of the distance computation, and thus to not consider the remaining dimensions. Super-EGO has been since improved by Gallet and Gowanlock [37], as part of a CPU-GPU distance similarity self-join algorithm. Among the changes, their version of Super-EGO is capable of FP64 computation while performing better than Super-EGO proposed by

Kalashnikov [57]. As such, further references to Super-EGO in this paper will refer to the work conducted by Gallet and Gowanlock [37], rather than Kalashnikov [57].

Perdacher et al. [82] propose FGF-Hilbert, a parallel CPU distance similarity join algorithm also based on an epsilon grid order, but using space-filling curves as their indexing method. Using an EGO-sorted dataset, space-filling curves are used to determine, for each query point, a range of consecutive candidate points in the dataset. The authors further improve the performance by using the OpenMP API and low-level vectorized instructions, making their solution highly optimized. Because FGF-Hilbert typically performs better than Super-EGO, particularly in higher dimensions, it is considered a state-of-the-art CPU distance similarity join algorithm. Because of some of its optimizations, FGF-Hilbert is only capable of FP64 computation.

Gowanlock and Karsin [44] propose GDS-Join, a GPU algorithm for high-dimensional distance similarity self-joins. Their optimizations related to the high-dimensional case include reordering the dimensions of the points based on their variance, so these with the highest variance would be considered first when computing distances. Similarly to Super-EGO presented above, this particularly pairs well with distance calculation short-circuiting. Overall, dimensions with a higher variance are susceptible to increase the cumulative distance more than dimensions with lower variance and are thus more likely to trigger short-circuiting the distance calculation. They also propose to index the data in fewer dimensions than the input dataset dimensionality, making their grid index efficient even in higher dimensions, as the cost of searching their grid index is bound by the number of dimensions that are indexed. Furthermore, as their source code is publicly available, it appears that new optimizations have been added to the GDS-Join algorithm since the first publication, including the use of Instruction-Level Parallelism (*ILP*) in the distance calculation, which significantly improves the performance of the algorithm. Our experiments show that this newer version of GDS-Join is more efficient than the published version [44]. Thus, we choose to use the newer more efficient version, as it is fairer than comparing TED-Join with the original algorithm.

5.3 Distance Calculations using Tensor Cores

We present our algorithm, TED-Join, that leverages TCs for Euclidean distance calculations, and show how it is integrated into a distance similarity self-join algorithm. For illustrative purposes, in this section we use 4 × 4 matrices; however using the WMMA API and FP64, matrix sizes are either 8 × 4, 4 × 8 or 8 × 8.

5.3.1 Adapting the Euclidean Distance Formula

Using the Euclidean distance formula defined above (Equation 5.1) between two points a and b in d dimensions, we can expand this formula as follows:

$$dist(a,b) = \sqrt{(a_d - b_d)^2 + \ldots + (a_1 - b_1)^2 + 0}. \tag{5.2}$$

We observe that, from right to left, the computation consists of a series of multiply-and-accumulate operations, where the distance in dimension i, computed as $(a_i - b_i)^2$ (hence a multiplication of two terms) gets accumulated with the distance previously computed in dimension $i-1$, where $1 < i < d$. Let a, b, c, d, e, f, g and h be eight points in d dimensions, and where we want to compute the Euclidean distance between a, b, c, d and e, f, g, h. For illustration purposes only, we will use 4 × 4 matrices.

$$A \quad \begin{bmatrix} a_1 & a_2 & a_3 & a_4 \\ a_1 & a_2 & a_3 & a_4 \\ a_1 & a_2 & a_3 & a_4 \\ a_1 & a_2 & a_3 & a_4 \end{bmatrix} \qquad B \quad \begin{bmatrix} e_1 & e_2 & e_3 & e_4 \\ f_1 & f_2 & f_3 & f_4 \\ g_1 & g_2 & g_3 & g_4 \\ h_1 & h_2 & h_3 & h_4 \end{bmatrix}$$

1. $B = B \times (-1.0)$ (CUDA cores)
2. $C = A \times I + B$ (TCs)
3. $D = C \times C^t + D$ (TCs)

Figure 5.1: Illustration of Euclidean distance calculations using TCs and Equation 5.2, between a point a and four points e, f, g, h, and in four dimensions. This computation is computed in blocking fashion four dimensions at a time. Matrix D contains the Euclidean distance between a and the other points.

We illustrate in Figure 5.1 how we can compute Euclidean distances using Equation 5.1

and more particularly its equivalent, Equation 5.2, using TCs. Matrix A contains a single point a stored in row-major, while matrix B can contain multiple points (here e, f, g, h), and is also row-major. To compute the difference between the coordinates, and to use TCs, we first scale B by a factor -1.0, and we compute $C = A \times I + B$, where I is the identity matrix. C thus contains the difference between all coordinates of a and the points e, f, g and h, and in all four dimensions (because matrices are 4×4). We then multiply C by its transpose, C^t, which computes the Euclidean distance between point a and the points e, f, g, h, in the current dimensions that we store in D. This calculation is computed in blocking fashion four dimensions at a time.

A severe limitation of using the Euclidean distance shown in Equation 5.1 and represented in Figure 5.1, is that it is only capable of computing the distance between one single point and several other points. Consider $D_{1,1}$ as the element in the first column of the first row of matrix D. The result of the computation in Figure 5.1 is that $D_{1,1} = dist(a, e)$, $D_{2,2} = dist(a, f)$, $D_{3,3} = dist(a, g)$ and $D_{4,4} = dist(a, h)$. Hence, out of the $4 \times 4 = 16$ results that matrix D can store, only 4 correspond to actual Euclidean distances. Thus, while TCs have a higher peak throughput than CUDA cores [73], only a fraction of the computation is actively used to compute Euclidean distances, which yields inefficient resource utilization. Furthermore, while we use 4×4 matrices for illustration purposes, we see in Figure 5.1 that all matrices used in the MMA operation need to have the same size, since the accumulator (C) is then used for the multiplication. However, when using the WMMA API and FP64, these matrix sizes are different and this solution can not be used. Consequently, we propose to use the expanded and equivalent form of the Euclidean distance outlined in Equation 5.1, which we detail as follows:

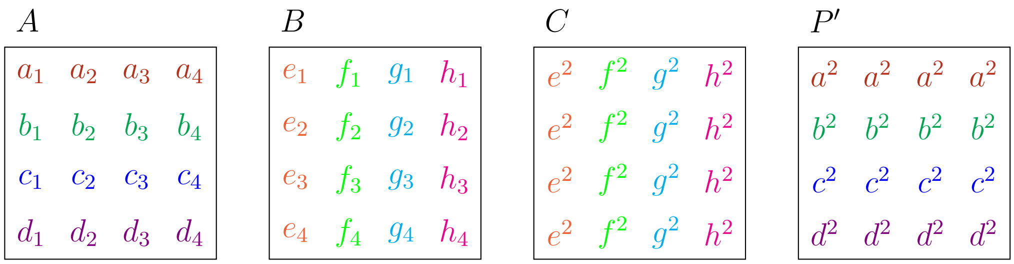
$$dist(a, b) = \sqrt{\sum_{i=1}^{d} a_i^2 - 2a_ib_i + b_i^2}. \tag{5.3}$$

Similarly to Equation 5.2, we can expand Equation 5.3, yielding the following equation:

$$dist(a,b) = \sqrt{\underbrace{a_d^2 + \overbrace{(-2a_db_d + b_d^2)}^{\text{Tensor cores}}}_{\text{CUDA cores}} + \ldots + \underbrace{a_1^2 + \overbrace{(-2a_1b_1 + b_1^2)}^{\text{Tensor cores}}}_{\text{CUDA cores}}}. \tag{5.4}$$

Using Equation 5.4, we emphasize which part of the computation will be carried out by TCs and which part by the CUDA cores. Let $T_i = -2a_ib_i + b_i^2$ be the MMA operation done by TCs. To compute $dist(a_i, b_i)$, we need to calculate $a_i^2 + T_i$. To use TCs, we need to transform this into an MMA operation, computing either $a_i^2 \times I + T_i$, or $T_i \times I + a_i^2$, where I is the identity matrix. However, as aforementioned, when using FP64 the WMMA API restricts us from reusing the accumulator from a previous MMA operation to be used in the multiplication of another MMA operation, due to different matrix sizes. Furthermore, using Equation 5.4, we can compute the Euclidean distance between the four points a, b, c, d, and the four other points e, f, g, h at a time, using the method illustrated in Figure 5.2, and which was not possible using Equation 5.1 (Figure 5.1). Finally, we observe that when computing the Euclidean distance between multiple points, and as will be the case when computing a distance similarity self-join for example, a part of the computation can be reused. The squared coordinates of the points (a_i^2 and b_i^2), are often reused throughout the computation. Indeed, the squared coordinates of a point are used for all the distance calculations with other points and do not change throughout the computation. Thus, the squared coordinates of the points can be precomputed to further improve the performance of the algorithm. As we still consider the use of 4×4 matrices for illustrative purposes, we store in an array P the squared and accumulated coordinates of each point, four coordinates at a time. Considering that a is the first point, the element 0 of this precomputed array P is $a_1^2 + a_2^2 + a_3^2 + a_4^2$. For a dataset V in d dimensions, this array represents a memory overhead of only $|V| \times \lceil d/4 \rceil$.

Figure 5.2 presents our algorithm design for computing the Euclidean distance between two sets of points, rather than between a single point and a set of points (Figure 5.1). This method is based on Equation 5.3. Matrix A contains the first set of points (a, b, c, d),



$$A = \begin{bmatrix} a_1 & a_2 & a_3 & a_4 \\ b_1 & b_2 & b_3 & b_4 \\ c_1 & c_2 & c_3 & c_4 \\ d_1 & d_2 & d_3 & d_4 \end{bmatrix} \quad B = \begin{bmatrix} e_1 & f_1 & g_1 & h_1 \\ e_2 & f_2 & g_2 & h_2 \\ e_3 & f_3 & g_3 & h_3 \\ e_4 & f_4 & g_4 & h_4 \end{bmatrix} \quad C = \begin{bmatrix} e^2 & f^2 & g^2 & h^2 \\ e^2 & f^2 & g^2 & h^2 \\ e^2 & f^2 & g^2 & h^2 \\ e^2 & f^2 & g^2 & h^2 \end{bmatrix} \quad P' = \begin{bmatrix} a^2 & a^2 & a^2 & a^2 \\ b^2 & b^2 & b^2 & b^2 \\ c^2 & c^2 & c^2 & c^2 \\ d^2 & d^2 & d^2 & d^2 \end{bmatrix}$$

1. $A = A \times (-2.0)$ (CUDA cores)
2. $T = A \times B + C$ (TCs)
3. $D = D + T + P$ (CUDA cores)

Figure 5.2: Illustration of Euclidean distance calculations using TCs and Equation 5.3, between four points a, b, c, d and four points e, f, g, h, and in four dimensions. This computation is computed in blocking fashion four dimensions at a time. Matrix D contains the Euclidean distances between a, b, c, d and e, f, g, h.

while B contains the second set of points (e, f, g, h). Matrix C contains the sum of squared coordinates of the points in B and are pre-computed, as explained above. Matrix P' contains the sum of squared coordinates of the points in A. Our algorithm first scales matrix A, and then computes $T = A \times B + C$ using TCs. We then use the CUDA cores to accumulate P', as well as the result matrix D. Because C, D, and T are different sizes than A and B, we can not use TCs to compute these operations, which is a limitation of the WMMA API when using FP64. This computation is computed in blocking fashion four dimensions at a time. The algorithm outputs matrix D which contains the Euclidean distance between a, b, c, d and e, f, g, h, which corresponds to 16 distances, compared to only 4 when using the algorithm shown in Figure 5.1. While we illustrate the computation using 4×4 matrices, when using the WMMA API, because D is an 8×8 matrix, we can compute 64 distances instead of 8.

5.3.2 Tensor Cores for Distance Similarity Joins

As we outlined in Section 5.2.4, most of the distance similarity self-join algorithms in the literature reduce the overall computational complexity by using an index data structure and, compared to a brute-force approach, typically reduces the number of candidate points that need to be refined per query point. In particular, the distance similarity self-join algorithm that we leverage here, GDS-JOIN, uses a grid index with cells of size ϵ^d. For each query point in the dataset V, we thus *search* the grid indexing for neighboring cells, yielding a set of candidate points for each of the query points, which are then *refined* by computing the Euclidean distance between them and the query point. Because TED-JOIN and GDS-JOIN use the same index, both algorithms yield the same candidate points to be refined using distance calculations. This allows us to compare the performance of CUDA and TCs in a self-consistent manner, where the performance differences are directly attributable to distance calculations.

A characteristic of the grid index we are using is that all the query points from the same cell share the same candidate points. This characteristic is particularly important, as it is necessary to efficiently make use of Equation 5.4 (Figure 5.2). Indeed, the query points we use in matrix A must compute their Euclidean distances, in matrix B, to the same set of candidate points. Hence, the query points used in matrix A should come from the same grid cell, as they share the same set of candidate points.

Another optimization used by Gowanlock and Karsin [44] is the batching of the execution. Because the final result of the similarity self-join might exceed the memory size of the GPU, the entire execution is split across multiple batches. As a positive side-effect, multiple batches allow for hiding data transfers between the host and the GPU with computation. Indeed, batches are computed by several parallel CUDA streams, where the data transfers of a stream can overlap the computation of another stream. However, as a batch corresponds to a set of query points to compute, we must ensure in our case that the query points we send in a batch can be computed by our TCs algorithm. More specifically, when assigning query points from a batch to a warp on the GPU, we must ensure that these query points belong in the same grid cell and are not from different cells. Otherwise, we would be unable to use the algorithm presented in Figure 5.2.

Using the WMMA API and FP64, only one combination of matrix sizes is available. Namely, matrix A will contain up to four coordinates of up to eight query points, matrix B up to four coordinates of up to eight candidate points, matrix C the sum of squared coordinates of up to eight candidate points, and matrix D up to sixty-four Euclidean distances. Because TCs operate at a warp level using the WMMA API, we assign up to eight query points to a warp, which will then compute the Euclidean distance to all the candidate points, as determined by the use of the grid index. If the number of query points, candidate points, or coordinates is insufficient to fill the remaining rows or columns of the matrices, we must fill them with zeros. Because we process four dimensions at a time, up to $\lceil d/4 \rceil$ steps are necessary to compute the Euclidean distance. Similarly to GDS-Join [44], we enable distance calculations short-circuiting, which may happen after every MMA operation, i.e., for every 4 dimensions. However, all currently computed Euclidean distances between all the query points and candidate points of the warp must short-circuit to trigger this optimization.

5.4 Experimental Evaluation

In this section, we detail the experimental evaluation we conducted. We start by comparing our TCs algorithm and another optimized TCs algorithm to compute Euclidean distances. We then compare our proposed algorithm TED-Join to other state-of-the-art distance similarity self-join algorithms.

5.4.1 Datasets

We evaluate the algorithms using a wide range of real-world and synthetic datasets, spanning several sizes, dimensionalities, and distributions. Synthetic datasets are generated following either a uniform or exponential distribution, and their name is prefixed by either *Unif* or *Expo*, respectively, followed by the dimensionality and the number of points (e.g., *Expo3D2M* is an exponentially distributed 3-D dataset containing 2M points). We summarize the different synthetic datasets that we use in Table 5.1, and the real-world datasets in Table 5.2. *Gaia50M* and *OSM50M* are the first 50M points of the original datasets, as described by Gowanlock [40]. We choose to use different distributions to better evaluate the

Table 5.1: Synthetic datasets used in the experimental evaluation.

Distribution	d	n
Uniform	2, 3, 4, 6, 8	10M
Exponential	2, 3, 4, 6, 8	2M, 10M

Table 5.2: Real-world datasets used in the experimental evaluation.

Dataset	d	n	Dataset	d	n
SW2DA [69]	2	1.86M	*SW2DB* [69]	2	5.16M
OSM50M [1]	2	50M	*Gaia50M* [2]	2	50M
SW3DA [69]	3	1.86M	*SuSy* [11]	18	5M
BigCross [3]	57	11M	*Songs* [18]	90	515K

performance of TCs under different workloads: when a dataset is uniformly distributed, TCs should all have a similar workload, while when a dataset is exponentially distributed, some TCs will have a higher workload than other TCs.

We denote the selectivity as s, which represents the average number of neighboring points found within ϵ of each query point when performing a similarity self-join, excluding each query point finding itself. The selectivity is calculated as follows: $s = (|R| - |V|)/|V|$, where $|R|$ and $|V|$ are the result set of the similarity self-join and dataset sizes, respectively. This metric is used in the literature to quantify the complexity of the search for a given value of ϵ: increasing ϵ results in more work to compute, and a higher selectivity.

5.4.2 Methodology

We conducted our experiments on the following platforms: **Platform 1**: 2× AMD Epyc 7542 CPU (2×32 cores, 2.9GHz), 512 GiB of RAM, Nvidia A100 GPU; **Platform 2:** Intel Xeon W-2295 CPU (18 cores, 3GHz), 256 GiB of RAM.

In this section, we use the distance similarity join application as a case study for the use of TCs for Euclidean distance calculations. For completeness, we compare our algorithm to other distance similarity join algorithms, including parallel CPU algorithms. However, this is only one example application, and thus we also show brute-force CUDA vs. TC performance as it may be more applicable to other algorithms.

The algorithms TED-Join, GDS-Join, Super-EGO and FGF-Hilbert are configured as follows:

- TED-Join: Our proposed TCs algorithm is executed on Platform 1, configured with 256 threads per block (8 warps), up to 8 query points per warp, and using distance calculations short-circuiting, as explained in Section 5.3.2.
- GDS-Join: Parallel GPU algorithm proposed by Gowanlock and Karsin [44] and further optimized since the original publication, executed in Platform 1. This algorithm is configured with 256 threads per block, $ILP = min(8, d)$ and uses distance calculations short-circuiting, as presented in Section 5.2.4.
- Super-EGO: Parallel CPU algorithm proposed by Kalashnikov [57], optimized by Gallet and Gowanlock [37] and executed on Platform 1 using 64 threads (the number of physical cores on the platform).
- FGF-Hilbert: Parallel CPU algorithm proposed by Perdacher et al. [82], executed on Platform 2 (the only platform supporting AVX-512, required for this algorithm) and using 18 threads (the number of physical cores on the platform).

While we would have preferred to use a single platform to conduct all our experiments, and thus have the same number of threads/cores for all CPU algorithms, prior experiments we conducted showed us that both Super-EGO and FGF-Hilbert had a relatively poor scalability. Hence, if we were able to run FGF-Hilbert using 64 threads/cores, as we did for Super-EGO, the results we show in the following sections would not have been significantly different. Furthermore, note that despite using fewer threads/cores, FGF-Hilbert typically outperforms Super-EGO.

All the algorithms are using double precision (FP64) to compute, and are compiled using NVCC v11.2 (for TED-Join and GDS-Join) or GCC (v8.5 for Super-EGO, and v9.4 FGF-Hilbert) using the O3 optimization.

During our experiments, many scenarios using FGF-Hilbert did not produce the correct self-join results, which are consequently not included. We believe that the issues encountered with FGF-Hilbert are due to the width of the vectorized instructions: 512-bits, or 8 FP64 values, which may not be working when $d < 8$. Furthermore, Super-EGO happened to fail in several low-dimensional cases without a clear understanding of the reason, and we thus also do not report the execution time of these experiments. However, we consider that the successful experiments should be sufficient to accurately evaluate the performance of TED-Join compared to the other algorithms. Finally, note that the four algorithms, TED-Join[2], GDS-Join [44], Super-EGO [57], and FGF-Hilbert [82], are publicly available.

5.4.3 Results: Comparison of Brute-force TC Approaches

We compare the performance of TCs and CUDA cores for performing Euclidean distance calculations when using brute-force computation, which is $O(|V|^2)$. Here, we use the algorithm TED-Join (TCs), to which we removed all optimizations, including indexing, and compare it to a highly optimized MMA reference implementation by Nvidia [77], that leverages the WMMA API similarly to TED-Join, denoted as WMMA-Ref. We selected this implementation instead of a library such as cuBLAS [3] or CUTLASS [4] (with the latter built upon the WMMA API), as it is the best direct comparison between approaches.

We outline two major differences between WMMA-Ref and TED-Join, as a consequence of matrix size, as follows:

1. The matrix sizes are dependent on data dimensionality and impact performance [84, 86]. WMMA-Ref is designed and optimized for large MMA operations, whereas TED-Join targets smaller matrices.

2. TED-Join uses small matrices, and thus computes many small 8×8 distance matrices and leverages shared memory. In contrast, WMMA-Ref computes the entire $|D|^2$

[2] `https://github.com/benoitgallet/ted-join-hipc22`

[3] `https://developer.nvidia.com/cublas`

[4] `https://github.com/NVIDIA/cutlass`

distance matrix, thus requiring a much larger memory footprint. Consequently, when using WMMA-Ref, and to be able to use it on large datasets that would exceed global memory capacity, we store the result matrix using unified memory, which automatically pages data between main and global memory. Furthermore, as cuBLAS and CUTLASS work similarly to WMMA-Ref, they have the same drawback related to the use of unified memory.

Figure 5.3 plots the performance of TED-Join and WMMA-Ref using brute-force searches (i.e., without using an index) to compute Euclidean distance calculations on a 16-D exponentially distributed synthetic datasets, spanning 2^{11} to 2^{17} points (we omit datasets with other dimensionalities as we observed similar results). Note that 2^{18} points overflows main memory when using WMMA-Ref. We observe that the performance of WMMA-Ref degrades quicker than TED-Join as the dataset size increases. We attribute these results to the use of unified memory by WMMA-Ref, which is required to store the large result matrix ($|D| \times |D|$), and which is paged between GPU global and main memory when its size exceeds global memory capacity. In addition to the poor performance attributed to unified memory, using WMMA-Ref, which computes on large matrices and thus on the d dimensions of a dataset at a time, limits the use of several optimizations, which are explored in the following sections. Namely, this inhibits short-circuiting the distance calculations when the cumulative distance between points exceeds ϵ.

We profile TED-Join and WMMA-Ref on the 2^{17} points 16-D dataset (Figure 5.3). With this dataset size, unified memory needs to be paged between global and main memory throughout the execution. We measure that WMMA-Ref transfers 687.84 GB between the L1 and L2 caches, and 503.61 GB between the L2 cache and global memory. In comparison TED-Join transfers 558.57 GB and only 0.046 GB, respectively, as we rely on shared memory to store small (8×8) result matrices, rather than a large $|V|\times|V|$ matrix in global memory like WMMA-Ref. This results in lower L1 and L2 hit rates: 19.35% using WMMA-Ref vs. 50.32% using TED-Join for the L1 hit rate, and 72.45% vs. 99.99% for the L2 hit rate.

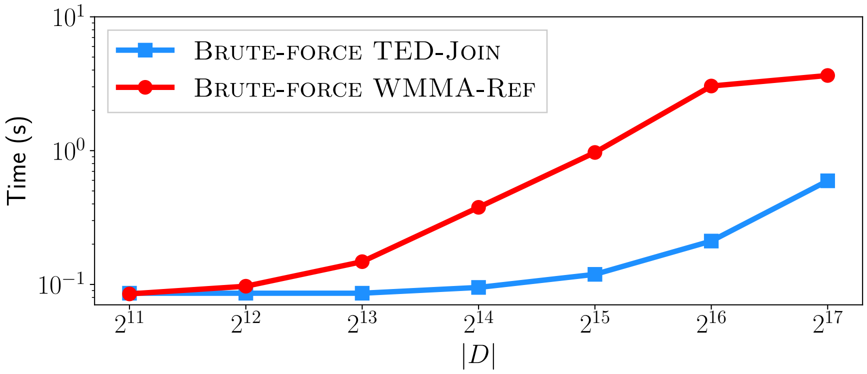


Figure 5.3: Response time of our proposed algorithm TED-Join, and WMMA-Ref an optimized MMA algorithm from Nvidia leveraging the WMMA API, using brute-force searches to compute Euclidean distance calculations on a 16-D exponentially distributed synthetic datasets.

In summary, the unified memory required by WMMA-Ref negatively affects performance in the case of distance calculations, and thus TED-Join should be preferred.

5.4.4 Results: Optimized TC and CUDA Core Approaches

We investigate in this section the performance of TED-Join, as compared to other state-of-the-art algorithms from the literature: GDS-Join, Super-EGO, and FGF-Hilbert.

5.4.4.1 Uniformly Distributed Datasets

We start this result section with uniformly distributed synthetic datasets, detailed in Table 5.1. We select this distribution as all the query points will have a similar number of candidate points to refine, allowing us to evaluate the performance of TCs when their workload is relatively uniform.

We show in Figure 5.4 the execution time of TED-Join compared to GDS-Join, Super-EGO, and FGF-Hilbert on a selection of uniformly distributed synthetic datasets. In these cases, we can see that Super-EGO is consistently performing worse than all of the other algorithms, except on the *Unif8D10M* dataset when $\epsilon = 0.08$ (Figure 5.4(d)). Furthermore, we observe that TED-Join performs similarly or better than GDS-Join in most

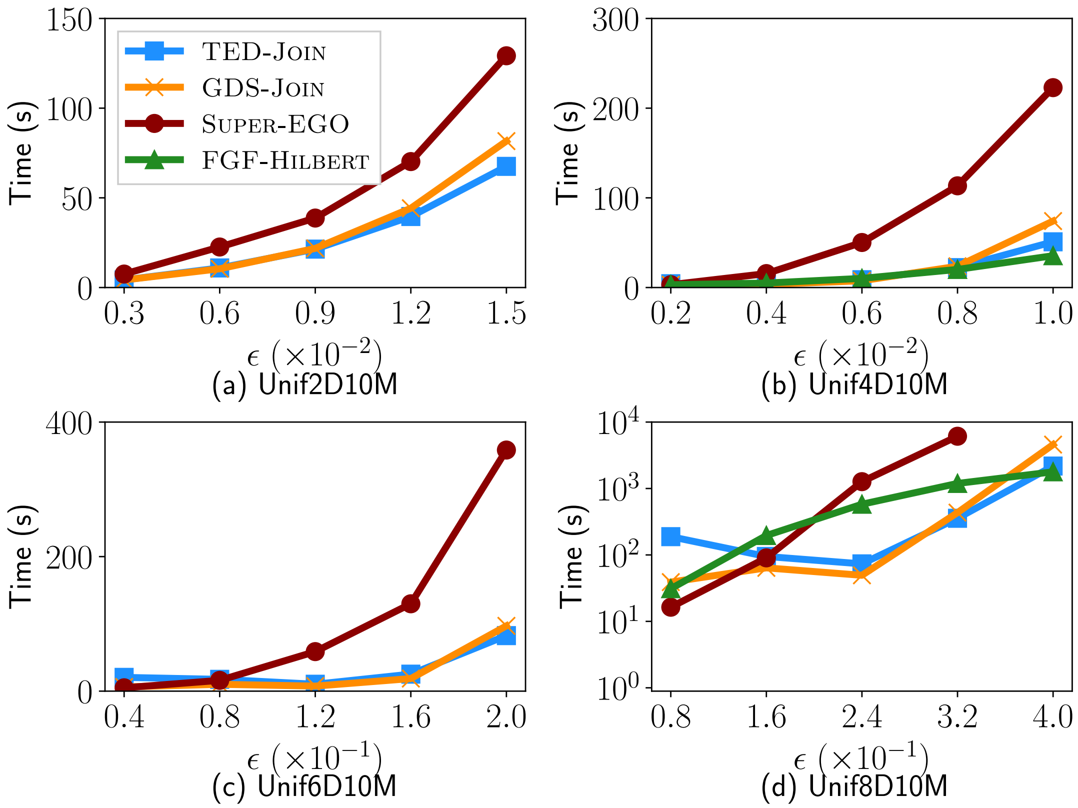


Figure 5.4: Response times of the TED-Join, GDS-Join, Super-EGO, and FGF-Hilbert on a selection of uniformly distributed synthetic datasets. s is in the range (a) 282–6978, (b) 71–8449, (c) 7–4295 and (d) 0–10888. The legend in (a) corresponds to all subfigures. $d \in \{2, 4, 6, 8\}$, $n = 10$M.

cases, except on *Unif8D10M* when $\epsilon < 0.32$. From these results, it seems that TED-Join performs similar to GDS-Join when ϵ is low, and therefore when the workload is low as well, potentially indicating an overhead from using TCs. But when ϵ increases, and thus the workload, the higher computational throughput of TCs outperforms the CUDA cores used by GDS-Join.

We also observe that the speedup is the highest on the 2-D and 4-D datasets since all 2 or 4 dimensions can be computed at once using TCs, as we compute 4 dimensions at a time. The speedup is the lowest on the 6-D datasets since we need to compute the distances in two iterations (as many as for the 8-D datasets), but where 2 dimensions are zeros and thus that the CUDA cores in GDS-Join do not have to compute.

5.4.4.2 Exponentially Distributed Datasets

In this section we present the results on the same algorithms as in Section 5.4.4.1 on the exponentially distributed synthetic datasets, detailed in Table 5.1. We select this distribution as it creates a large workload variance between the query points, where some query points may have many candidate points to refine, and other query points very few, which allows us to evaluate the performance of the TCs when their workload varies.

Figure 5.5 reports the execution time of TED-Join compared to GDS-Join, Super-EGO, and FGF-Hilbert on a selection of exponentially distributed synthetic datasets. Note that FGF-Hilbert did not run correctly on the 2-D and 6-D datasets (Figures 5.5(a) and (c)). In these experiments, TED-Join typically performs similarly or better than GDS-Join, particularly as ϵ increases. Super-EGO is consistently outperformed by the other algorithms, while FGF-Hilbert performs the best on the *Expo4D10M* dataset (Figure 5.5(b)), but is outperformed by both TED-Join and GDS-Join on the *Expo8D10M* dataset (Figure 5.5(d)). Because these datasets are exponentially distributed, the workload throughout the computation of the similarity self-join can vary a lot. The query points in the denser regions of the dataset will have many candidate points to refine, and the query points in the sparse regions of the dataset may have only a few candidate points. Hence, and despite a highly varying workload, TED-Join remains more efficient in most cases compared to GDS-Join and all compared algorithms in general, particularly in lower dimensions ($2 \leq d \leq 4$).

5.4.4.3 Real-World Datasets

We present in this section the results of TED-Join, GDS-Join, Super-EGO and FGF-Hilbert on a selection of the real-world datasets (Table 5.2), as shown in Figure 5.6. TED-Join and GDS-Join perform very similarly, particularly on the higher dimensional datasets (Figures 5.6(b)–(d)), while TED-Join outperforms GDS-Join on the *SW3DA* dataset as ϵ increases (Figure 5.6(a)). FGF-Hilbert also performs quite similarly to TED-

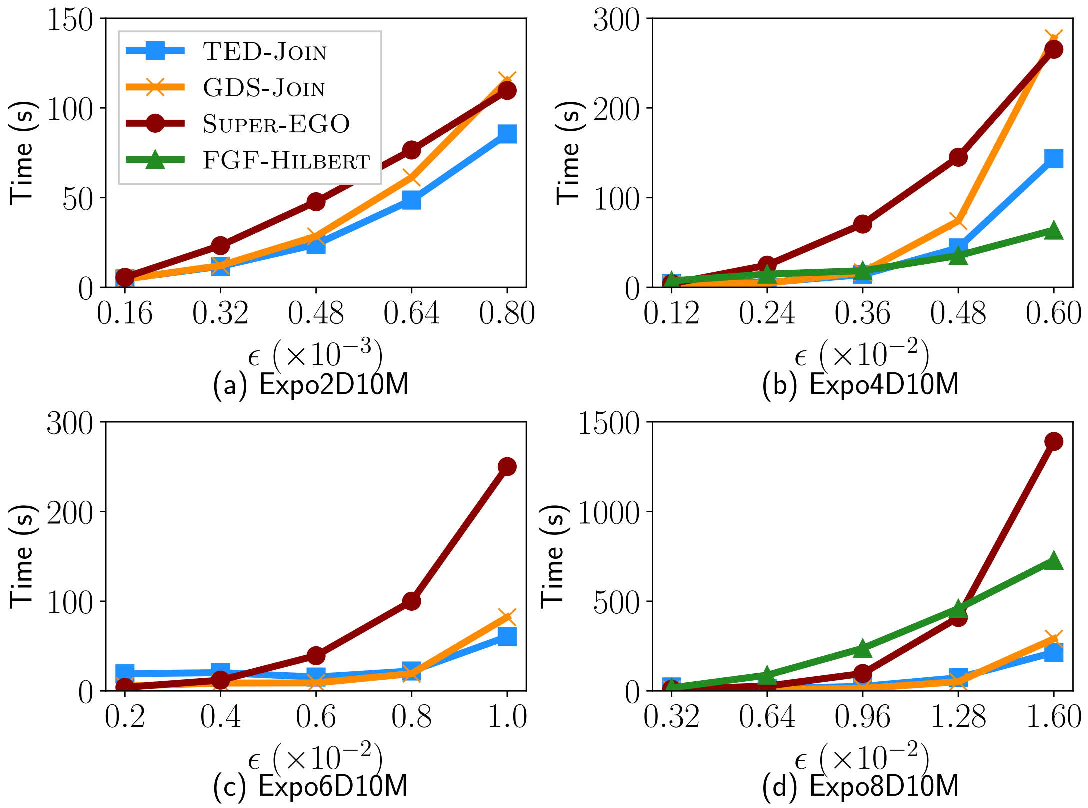
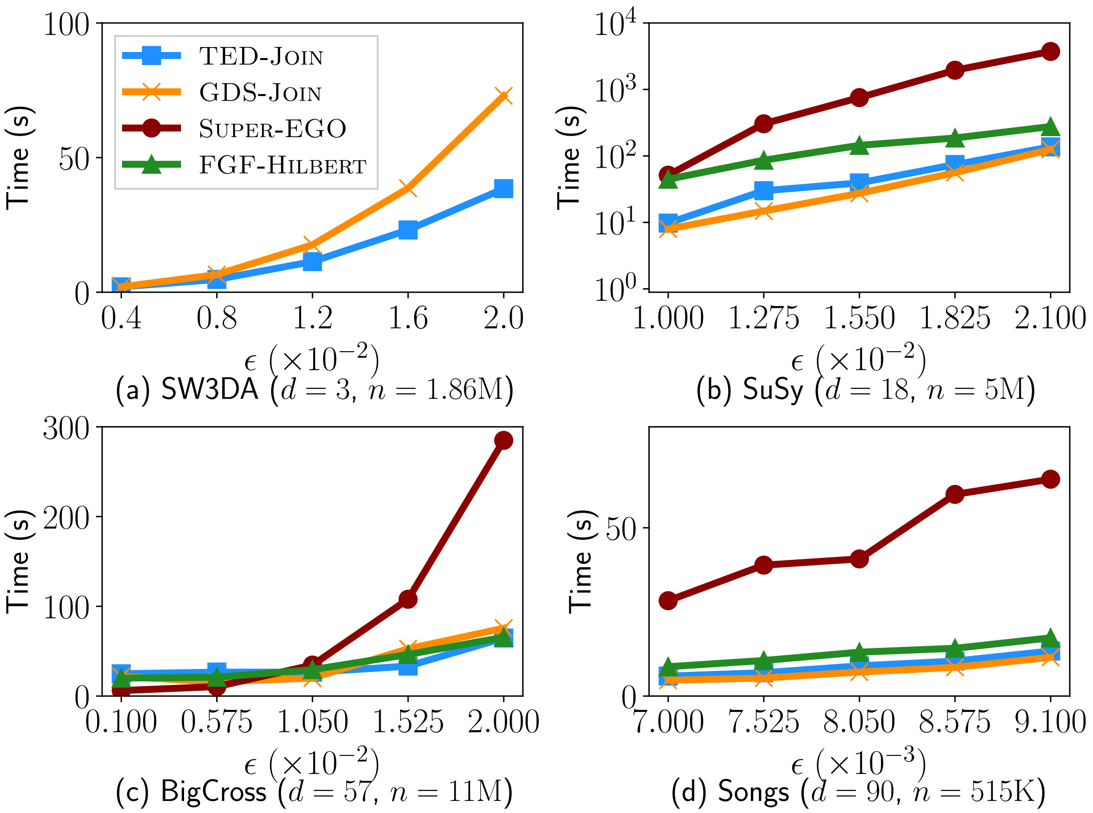


Figure 5.5: Response times of TED-Join, GDS-Join, Super-EGO, and FGF-Hilbert on a selection of exponentially distributed synthetic datasets. s is in the range (a) 320–7834, (b) 15–7414, (c) 0–1658 and (d) 0–1210. The legend in (a) corresponds to all subfigures. $d \in \{2, 4, 6, 8\}$, $n = 10$M.

Join and GDS-Join, while Super-EGO is often outperformed by the other algorithms. Overall, these experiments show that TED-Join and GDS-Join may perform similarly as dimensionality increases, while TED-Join yields an advantage in lower dimensions (Figure 5.6(a)), as we observed in previous Figures 5.4 and 5.5. These experiments show us that in higher dimensions (Figures 5.6(b)–(d)), TED-Join may not yield an advantage compared to GDS-Join.

5.4.5 Discussion: When Tensor Cores Should Be Employed

We summarize the results of TED-Join as compared to the Super-EGO [37], FGF-Hilbert [82], and GDS-Join [44] algorithms that we obtained across experiments, including those that were omitted due to space constraints. The experiments covered a wide range of



(a) SW3DA ($d = 3$, $n = 1.86$M)

(b) SuSy ($d = 18$, $n = 5$M)

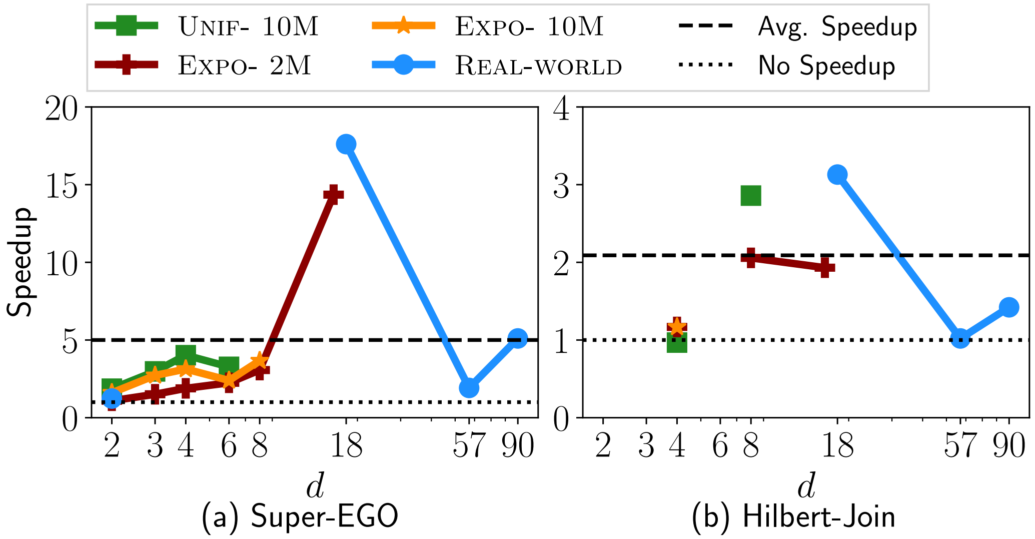
(c) BigCross ($d = 57$, $n = 11$M)

(d) Songs ($d = 90$, $n = 515$K)

Figure 5.6: Response times of TED-Join, GDS-Join, Super-EGO, and FGF-Hilbert on a selection of real-world datasets (Table 5.2). s is in the range (a) 163–5373, (b) 5–1090, (c) 1–1104 and (d) 127–998. The legend in (a) corresponds to all subfigures.

data dimensionalities, sizes, and distributions, resulting in an insightful picture of the overall performance of using TCs in TED-Join compared to the use of CUDA cores in GDS-Join. We report the speedup of TED-Join over the Super-EGO, FGF-Hilbert, and GDS-Join algorithms in Figures 5.7(a) and (b), and Figure 5.8, respectively. We also report the L1 and L2 cache hit rates of GDS-Join and TED-Join in Table 5.3, and the average and maximum speedups of TED-Join over Super-EGO, FGF-Hilbert, and GDS-Join in Table 5.4.

Figure 5.7(a) plots the speedup of TED-Join over the CPU algorithm Super-EGO [37, 57]. We observe that TED-Join consistently achieves a speedup > 1, with an average of 5.00× and a maximum of 27.22×. Thus, we believe that there is no clear disadvantage to using TED-Join over Super-EGO, regardless of the dimensionality, dataset distribution,



(a) Super-EGO

(b) Hilbert-Join

Figure 5.7: Speedups of TED-Join over (a) Super-EGO and (b) FGF-Hilbert across datasets presented in Tables 5.1 and 5.2, for all values of ϵ we used, and as a function of the dimensionality. The dashed horizontal lines correspond to the average speedups of TED-Join over a compared algorithm, and the dotted horizontal lines represent no speedup.

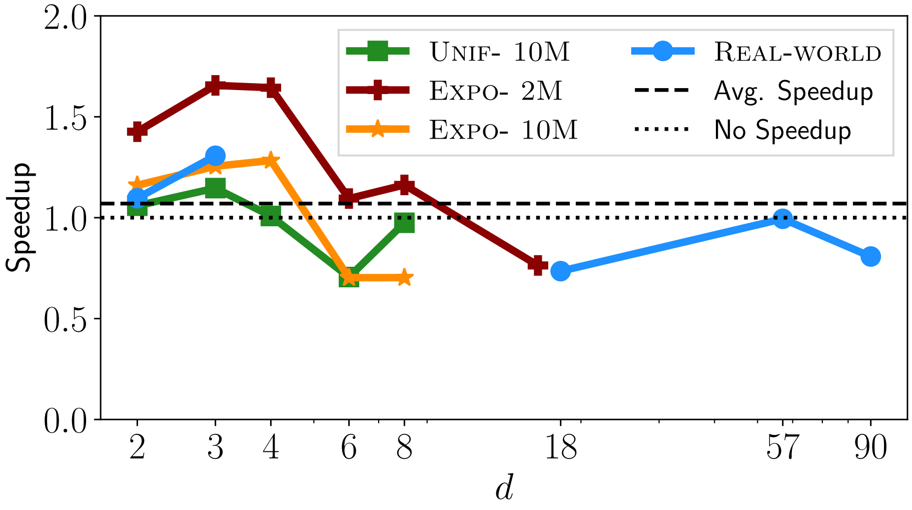


Figure 5.8: The same as for Figure 5.7, but plotting the speedup of TED-Join over GDS-Join.

or size.

Figure 5.7(b) plots the speedup of TED-Join over the CPU algorithm FGF-Hilbert [82]. Because many of our experiments could not be correctly conducted using the FGF-Hilbert algorithm, it makes it harder to draw a clear conclusion regarding the performance TED-

Table 5.3: L1 and L2 cache hit rates of GDS-Join and TED-Join on a selection of exponentially distributed synthetic datasets ($2 \leq d \leq 16$, $n = 2$M) and real-world datasets (SW3DA and SuSy), measured using the Nvidia Nsight Compute profiler.

	GDS-Join		TED-Join	
Dataset	L1	L2	L1	L2
Expo2D2M	71.55%	97.85%	66.40%	98.11%
Expo4D2M	89.89%	95.60%	67.20%	97.65%
Expo8D2M	90.53%	97.15%	45.76%	66.23%
Expo16D2M	97.27%	99.84%	52.15%	57.27%
SW3DA	68.84%	97.62%	54.70%	94.43%
SuSy	92.13%	86.91%	38.00%	53.50%

Table 5.4: Average and maximum speedup of TED-Join over Super-EGO, FGF-Hilbert, and GDS-Join across experiments reported in Figures 5.7 and 5.8.

	CPU		GPU
	Super-EGO	FGF-Hilbert	GDS-Join ($d \leq 4$)
Average	5.00×	2.09×	1.07× (1.28×)
Maximum	27.22×	9.46×	2.23× (2.23×)

Join compared to FGF-Hilbert. However, in the successful experiments, our TCs solution achieved an average speedup of 2.09× with a maximum of 9.46×, and the majority of the speedups are above 1. Hence, and similarly to Super-EGO, there is no clear disadvantage of using TED-Join over FGF-Hilbert.

Observing the speedup of TED-Join over the CUDA core algorithm GDS-Join (Figure 5.8), we achieve the best performance when $d \leq 4$, and is best on exponentially distributed synthetic and real-world datasets. However, as the dimensionality d increases, this speedup decreases, resulting in an average speedup of only 1.07×, but achieving a maximum of 2.23× on the *Expo3D2M* dataset. If we only consider datasets where $d \leq 4$, TED-Join achieves an average speedup of 1.28× over GDS-Join. Regarding the relatively low speedup in higher dimensions, TCs are designed for large matrix multiplications, where data can be reused when computing tiles of the resulting matrix. In the case of TED-Join, we are unable to reuse such data, thus limiting the performance. Additionally, we measure and compare the L1 and L2 cache hit rates of GDS-Join and TED-Join (Table 5.3). While

GDS-Join consistently achieves high cache hit rates, as the dimensionality increases, the cache hit rate using TED-Join decreases significantly. This explains why the speedup of TED-Join over GDS-Join decreases with increasing dimensionality (Figure 5.8).

From these results, we conclude that TCs should be used when the dimensionality is low ($2 \leq d \leq 4$). Furthermore, there are cases where the dimensionality does not evenly divide by 4 (the dimension of the matrices as defined by the WMMA API for FP64). In total, $\lceil d/4 \rceil$ MMA operations are needed to compute distance calculations, meaning that an additional MMA operation needs to be performed for cases where $d \bmod 4 \neq 0$, which performs excess work. For example, because 6-D datasets are stored as 8-D datasets, where the last two dimensions are filled with zeros, TCs cannot achieve peak performance.

In summary, TCs should be used under the following scenarios instead of the reference implementations on their respective architectures:

- Compared to using CUDA cores, TCs should be used on low-dimensional datasets ($2 \leq d \leq 4$).
- There is no drawback of using TCs over multi-core CPUs.

5.5 Conclusion and Future Work

In this paper, we presented a novel approach to computing Euclidean distances leveraging TCs on Nvidia GPUs. TCs are designed solely for Matrix Multiply-Accumulate operations, and yield a much higher peak throughput than CUDA cores for this operation [73]. While TCs have been extensively used in fields such as machine learning, their usage remains very limited for more general-purpose applications. Hence, to our knowledge, this paper presents the first use of TCs for FP64 Euclidean distance calculations, where FP64 TCs computation has only been possible using the Ampere generation of Nvidia GPUs. This makes our algorithm suitable for scenarios where precise computation using FP64 is required. As such, our algorithm can provide the foundation for improving the performance of other data analysis applications where distance calculations are used (e.g., distance similarity searches, kNN, and DBSCAN [32, 37, 41, 44, 57, 82]). In these cases, our TC GPU kernel can be adapted to refine candidate points independently of the index that is used.

Comparison to tensor algorithms: we compared TED-Join to a reference MMA implementation, WMMA-Ref, from Nvidia [77], where no optimizations (including an index) were used. We find that TED-Join outperforms WMMA-Ref, because the latter requires unified memory to store a $|V| \times |V|$ distance matrix. Libraries such as cuBLAS and CUTLASS have the same drawbacks as WMMA-Ref, and are thus also unsuitable for moderately sized input datasets.

Comparison to similarity search reference implementations: we compared TED-Join to the GPU algorithm GDS-Join [44]. Despite an average speedup of 1.07× over GDS-Join when $2 \leq d \leq 90$, we achieve a maximum speedup of 2.23× over this algorithm. We find that TED-Join yields the best performance when $d \leq 4$ with an average speedup of 1.28× over GDS-Join. Because TED-Join and GDS-Join use the same index, this performance improvement is a direct result of employing TCs. While the maximum speedup is expected to be 2× due to the maximum throughput of TCs compared to CUDA cores [73], we achieve a lower speedup on average because we rely on operations using CUDA cores. As described in Section 5.3.1, combining CUDA and TCs to compute FP64 Euclidean distances is required due to the restricted matrix sizes when using the WMMA API and FP64.

Compared to the multi-core CPU algorithms Super-EGO [57] and FGF-Hilbert [82], we find that TED-Join typically outperforms these algorithms.

Future work includes, investigating cache and shared memory efficiency, particularly for higher dimensions, modeling TC performance to determine in which scenarios they should be leveraged instead of CUDA cores, using other floating point precisions available for TCs, and incorporating our TC GPU kernel into other algorithms, such as kNN [41], and particle simulations such as those in molecular dynamics [28].

Acknowledgements

This material is based upon work supported by the National Science Foundation under Grant No. 2042155.

Chapter 6

Optimizing Euclidean Distance Calculations in Low Precision with GPU Tensor Cores

Abstract

Euclidean distance calculations are used in data analysis algorithms and many scientific applications. As such, they are often a performance bottleneck in numerous algorithms and applications, and significant research has proposed methods to avoid performing distance calculations. GPU Tensor Cores (TCs) are purposefully designed to compute Matrix Multiplication and Accumulation (MMA) operations and are much faster at this than general-purpose GPU cores. TCs have been extensively used in machine learning and other related fields; however, they have not been extensively used in other general-purpose algorithms. In this paper, we present a Euclidean distance algorithm that leverages TCs. Our TC algorithm uses low precision, and can thus take advantage of TCs that are commonly found on consumer-grade and high-end GPUs. In particular, we investigate half precision multiplication and half or single precision accumulation. We observe that accumulating in half precision instead of single precision slightly improves performance while keeping a high level of accuracy. Furthermore, we observe that in a real-world application that uses Euclidean distances, our TC algorithm consistently outperforms a state-of-the-art GPU algorithm using CUDA cores across a broad range of datasets, achieving a maximum speedup of 5.38× using single precision accumulations, and 6.45× when compared to double precision.

6.1 Introduction

Euclidean distance calculations are a fundamental operation in many data analysis algorithms such as k-means, nearest neighbor searches or clustering [22, 32, 57, 82]. These calculations are typically computationally intensive, especially when processing large datasets or those in high dimensions.

Graphics Processing Units (GPUs) have been extensively used to improve the performance of Euclidean distance calculations [22, 56]. Tensor Cores (TCs) are an Application-Specific Integrated Circuit (ASIC) available on some GPUs, and are explicitly designed to compute Matrix Multiplication and Accumulation (MMA) operations. Given four matrices A, B, C and D, TCs compute $D = A \times B + C$, where C and D can be the same matrix. Due to their high specificity, TCs yield a greater computational throughput computing MMA operations than general-purpose CUDA cores. Most of the literature uses TCs to improve the performance of machine learning applications, or more generally, applications requiring linear algebra kernels [61]. Nonetheless, any algorithm that can be expressed as MMA operations can leverage TCs to harness their power, with possible significant performance gains over general-purpose (CUDA) cores.

In this paper, we propose an algorithm to compute Euclidean distances using TCs. In particular, our algorithm focuses on lower precision data types, i.e., FP16 multiplications and FP16/FP32 accumulations. We use the Warp Matrix Multiply and Accumulate (WMMA) API [8] to leverage TCs. We evaluate our proposed algorithms in terms of execution time, but also in terms of accuracy. Finally, we demonstrate the performance of our algorithm in two contexts: (1) in a brute-force scenario that only computes Euclidean distances; and, (2) in an example algorithm, the distance similarity self-join, which is employed in many data analytic tasks. The distance similarity self-join algorithm finds all pairs of points, (a, b) from a given dataset V in d dimensions that are within a user-defined distance threshold ϵ of each other.

The paper most related to our work is that by Gallet and Gowanlock [38], which proposed an FP64 Euclidean distance algorithm leveraging TCs. Because their work does not explore lower precision data types, and because they report modest performance gains in higher dimensions compared to CUDA cores, we believe it is important to tackle these shortcomings. Additionally, their FP64 TCs algorithm is only capable of running on a very few number of GPU architectures [73, 76], whereas our solution can be executed on any Nvidia GPU equipped with TCs [71, 72, 73, 76]. Hence, we base a part of this work on the work conducted by Gallet and Gowanlock [38], which we further optimize. Furthermore, we propose a more in-depth evaluation of TCs, including a comparison of performance and accuracy. For consistency, we use the same notation and several concepts regarding TCs as in their paper. Thus, we direct the reader to their paper for additional details on computing Euclidean distances with TCs. We summarize our contributions as follows:

- We propose Mixed-precision Euclidean Tensor cores Algorithm (META), a novel algorithms that optimizes Euclidean distance computations using low precision data types (FP16 and FP32) on GPU TCs.

- We propose two variants of META: META-1Q, capable of computing the distance between 1 and 16 points; and META-16Q, capable of computing the Euclidean distance between 16 points and 16 other points per warp. META-1Q only makes use of TCs, while META-16Q uses a mix of CUDA and TCs to further improve the performance of the algorithm.

- We optimize the work conducted by Gallet and Gowanlock [38] by reordering the input dataset, greatly improving the memory accesses efficiency and overall performance.

- We show the performance of using META to compute Euclidean distances compared to using CUDA cores, and we also demonstrate the performance of using META when included in a distance similarity self-join algorithm and compare it to the performance of a state-of-the-art algorithm that uses CUDA cores. We find that META consistently outperforms the CUDA core algorithm on a broad range of datasets spanning different distributions, sizes, and dimensionalities.

- We measure the accuracy of META compared to CUDA cores. We observe the phenomenon of catastrophic cancellation [26, 39], which is a loss of precision occurring in some floating point operations and yielding incorrect results.

The paper is organized as follows: we present necessary background information in Section 6.2. We present in Section 6.3 our main algorithm, META, and its two variants: META-1Q and META-16Q, as well as their integration into the distance similarity self-join algorithm. We evaluate the performance and accuracy of META in Section 6.4, then conclude the paper and propose future research paths in Section 6.5.

6.2 Background

6.2.1 Problem Statement

Let a and b be two points in d dimensions, where a_i is the i^{th} coordinate of point a, where $i = 1, \ldots, d$. The Euclidean distance between a and b is commonly defined as follows:

$$dist(a, b) = \sqrt{\sum_{i=1}^{d} (a_i - b_i)^2}. \tag{6.1}$$

Additionally, Gallet and Gowanlock [38] redefined the Euclidean distance calculation such that it can be computed on TCs, as follows:

$$dist(a, b) = \sqrt{\sum_{i=1}^{d} a_i^2 - 2a_ib_i + b_i^2}. \tag{6.2}$$

6.2.1.1 Distance Similarity Self-join

We take the distance similarity self-join as a case-study application for our Euclidean distance algorithm. Given a dataset V in d dimensions, the distance similarity self-join algorithm finds all the pairs of points (a, b), where $a, b \in V$, and where $dist(a, b) \leq \epsilon$. In this paper, $dist()$ uses the Euclidean distance defined above as the measuring function. Given a query point a, finding all the points $\in V$ that are within ϵ from a is called a distance similarity search. By default, to find all points within ϵ from a, we calculate the distance between a and all the other points in V, yielding an overall complexity of $O(|V|)$.

6.2.2 Tensor Cores (TCs)

GPU TCs are an Application-Specific Integrated Circuit (ASIC), that are specifically designed to compute Matrix Multiplication and Accumulation (MMA) operations. TCs are made to compute the operation $D = A \times B + C$, where A, B, C and D are matrices, and where the accumulator matrices C and D may be the same. In this paper, we choose to use the WMMA API [8] from Nvidia to leverage TCs. This API yields programmatic access to matrix fragments, which represents matrices in memory and store their content in registers accessible to threads in a warp. In particular, the API provides several fragment-related functions, including filling a fragment with a given value, copying data from global memory to a fragment and vice versa, and computing an MMA operation. Compared to (general purpose) CUDA cores, TCs have a much greater computational throughput when computing MMA operations. For instance, using an Nvidia A100 GPU [73] and FP16 computation, TCs have a theoretical computational throughput 4× greater than CUDA cores.

Focusing on low precision Euclidean distance calculations, we use TCs to compute the multiplication step of the MMA operation using FP16, and the accumulation step using either FP16 or FP32. We will compare the performance and accuracy of both options in the experimental evaluation section of this paper (Section 6.4). Using the WMMA API, the available matrix sizes are either: 16×16 for all matrices A, B, C and D, or 8×32 for either

A or B, and 8×8 for C and D. In this paper, we elect to use 16×16 matrices, as it is required for our META-1Q algorithm.

6.2.3 Tensor Cores in the Literature

As mentioned in Section 6.1, the literature on TCs is heavily centered on machine learning or, more generally, linear algebra [61], but rarely other types of computation [4, 27, 56, 63, 67]. We highlight in this section a few papers leveraging TCs for more general-purpose computations, similar to what we are doing in this paper.

Dakkak et al. [27] leverage TCs to compute reductions and scans using the WMMA API. The reduction algorithm multiplies a matrix containing ones in the first row and zeros in the remaining rows, with another matrix containing the values to reduce. This matrix multiplication is then accumulated with the matrix containing the result from previous reductions. The scan algorithm uses a triangular matrix that contains ones and the remainder are zeros, which works similarly to the reduction algorithm. Compared to state-of-the-art algorithms using TCs, the reduction algorithm achieves a speedup of 100×, and the scan algorithm a speedup of 3×.

Ji and Wang [56] propose a DBSCAN algorithm using TCs. TCs are essentially used to compute distance matrices between points that are likely to form a cluster. Major differences with this paper are that we use the Euclidean distance, whereas Ji and Wang use the cosine similarity, and we use an index structure while Ji and Wang do not. TCs are used to compute reductions to determine if points belong to a cluster or not. Compared to CUDA cores, their solution using TCs achieves a speedup of up to 2.61×.

Ahle and Silvestri [4] propose a theoretical method to use TCs to compute similarity searches, using either the Hamming or squared L_2 distances or the cosine similarity. They also propose to use the Local Sensitivity Hashing (LSH) method to reduce the overall complexity, similar to using an index as used by other similarity search algorithms [38, 44, 57, 82]. However, LSH is typically an approximate method, contrary to the other algorithms aforementioned.

6.2.4 Case Study: Distance Similarity Searches

We discuss in this section several state-of-the-art parallel distance similarity self-join algorithms [37, 44, 57, 82] that use Euclidean distance calculations to compute the similarity searches. All of these algorithms use an indexing data structure to prune the number of distance calculations, which is a commonly used optimization [21, 22]. When using an index, it is first *searched* to yield a set of candidate points for each query point. The set of candidate points is then *refined* using distance calculations to keep pairs of query and candidate points that are within ϵ of each other. Note that we include CPU algorithms as a matter of exhaustiveness, despite not including them in the experimental evaluation. Indeed, these algorithms have been greatly outperformed by the other GPU algorithms, and are also not capable of computing using FP16, which is the topic of this paper.

6.2.4.1 Parallel CPU Algorithms

Kalashnikov [57] proposes the parallel CPU algorithm SUPER-EGO, which computes distance similarity joins. The performance of this algorithm relies on a grid index with ϵ-wide grid cells, that allow to efficiently prune the candidate points to refine. Furthermore, the features of the points are reordered based on their decreasing variance which, when computing the distance between the points, allows to short-circuit the computation and not consider all dimensionalities. While still considered a state-of-the-art CPU parallel algorithm for distance similarity joins, it is not capable to compute using FP16 data types, and we will thus not include it in our experiments.

Perdacher et al. [82] propose another parallel CPU algorithm, FGF-HILBERT, to compute distance similarity joins. They use a dataset sorted based on the points coordinates and space-filling curves are used as an index to determine, for each query point, a range of consecutive candidate points to refine. The algorithm is parallelized using OpenMP, as well as low-level vectorized instructions. FGF-Hilbert is a state-of-the-art parallel CPU algorithm as well, which is also not capable to compute using FP16 data types. FGF-Hilbert will thus not be included in our experiments.

6.2.4.2 GPU Algorithms

Gowanlock and Karsin [44] propose the state-of-the-art distance similarity self-join GPU algorithm GDS-Join, which uses CUDA cores. GDS-Join is a high-dimensional distance similarity self-join algorithm that uses CUDA cores, and was shown to consistently outperform Super-EGO and FGF-Hilbert. They use similar dimension reordering and short-circuiting techniques as Super-EGO, and propose to index fewer dimensions than the dataset's dimensionality d. This technique helps avoid the curse of dimensionality, where searching an index is exponentially more expensive as the number of indexed dimensions increases. GDS-Join also uses Instruction-Level Parallelism (ILP) to further improve the performance of Euclidean distance calculations. We modify this algorithm to make it capable to compute using FP16 and mixed FP16-FP32 precisions.

6.2.5 TCs Distance Similarity Searches

Gallet and Gowanlock [38] propose TED-Join, a GPU algorithm that uses TCs to compute Euclidean distances, and that they incorporate into GDS-Join to create a distance similarity self-join algorithm using TCs instead of CUDA cores. As such, their algorithm is very similar to GDS-Join presented above. They compare their algorithm to Super-EGO and FGF-Hilbert, as well as to GDS-Join. While TED-Join consistently outperforms Super-EGO and FGF-Hilbert, TED-Join outperforms GDS-Join only when the dimensionality is low ($d \leq 4$), making TED-Join a low-dimensional algorithm. This algorithm is the first to propose using TCs to compute Euclidean distances, and among the first papers to use TCs for FP64 general-purpose computations. However, their algorithm exclusively uses FP64, leaving FP16/32 unexplored. As such, we base our algorithm META upon their

$$\begin{array}{lllll} A & B & A' & P & Q \\ \begin{bmatrix} a_1 & a_2 & a_3 & a_4 \\ a_1 & a_2 & a_3 & a_4 \\ a_1 & a_2 & a_3 & a_4 \\ a_1 & a_2 & a_3 & a_4 \end{bmatrix} & \begin{bmatrix} e_1 & e_2 & e_3 & e_4 \\ f_1 & f_2 & f_3 & f_4 \\ g_1 & g_2 & g_3 & g_4 \\ h_1 & h_2 & h_3 & h_4 \end{bmatrix} & \begin{bmatrix} a_1 & a_2 & a_3 & a_4 \\ b_1 & b_2 & b_3 & b_4 \\ c_1 & c_2 & c_3 & c_4 \\ d_1 & d_2 & d_3 & d_4 \end{bmatrix} & \begin{bmatrix} e^2 & f^2 & g^2 & h^2 \\ e^2 & f^2 & g^2 & h^2 \\ e^2 & f^2 & g^2 & h^2 \\ e^2 & f^2 & g^2 & h^2 \end{bmatrix} & \begin{bmatrix} a^2 & a^2 & a^2 & a^2 \\ b^2 & b^2 & b^2 & b^2 \\ c^2 & c^2 & c^2 & c^2 \\ d^2 & d^2 & d^2 & d^2 \end{bmatrix} \end{array}$$

META-1Q:

1. $B = B \times (-1.0)$ (CUDA)
2. $C = A \times I + B$ (TCs)
3. $D = C \times C^t + D$ (TCs)

META-16Q:

1. $A' = A' \times (-2.0)$ (CUDA)
2. $T = A' \times B^t + Q$ (TCs)
3. $D = D + T + P$ (CUDA)

Figure 6.1: Illustration of Euclidean distance calculations for the META-1Q and META-16Q algorithms presented below (Sections 6.3.1 and 6.3.2). Reproduced and modified from Gallet and Gowanlock [38]. While we show 4 × 4 matrices for illustration purposes, META-1Q and META-16Q use 16 × 16 matrices.

work, which we propose to extend in this paper.

6.3 Mixed-Precision Euclidean Tensor cores Algorithm (META)

We present in this section the two different variants that compose META: META-1Q, capable of computing Euclidean distances between 1 and 16 points; and META-16Q, capable of computing Euclidean distances between 16 and 16 other points, in 16 dimensions at a time in both cases. For illustrative purposes, we will show in this section 4×4 matrices, while META uses 16×16 matrices. Part of this work is based upon the TCs algorithm (TED-Join) proposed by Gallet and Gowanlock [38]. We expand the scope of their work by addressing Euclidean distances as computed using TCs supporting lower floating point precision in addition to optimizing the data layout of their algorithm. For consistency, we use the same notation as in their paper [38].

6.3.1 META-1Q Algorithm

The first variant of META that we propose is META-1Q, which uses the common Euclidean distance formula given in Equation 6.1. META-1Q requires square matrices; thus, Gallet and Gowanlock [38] were not able to provide a similar algorithm using FP64 because of the matrix size restrictions [78]. We illustrate the META-1Q algorithm in Figure 6.1.

As mentioned above, TCs can only compute $D = A \times B + C$, whereas the formula in Equation 6.1 consists of a difference first. We first scale matrix C by a factor -1.0 and use matrix A or B as the identity matrix. Thus, the MMA operation will consist of computing $A - C$ or $B - C$. Let a, b, c, d, e, f, g and h be eight points in d dimensions, and where we want to compute the Euclidean distance between a, b, c, d and e, f, g, h. Because of the difference in the computation, that is applied on pair-wise coordinates, only 1 point from one set and the 4 points from the other set can be used at a time. We use matrix D to store the results, and we first fill matrix A with 4 coordinates of the point a in row-major order, replicated across each row of the matrix. We set matrix B as the identity matrix, and we fill matrix C with 4 coordinates of points e, f, g, h in row-major order, which we then scale by a factor -1.0, and compute an MMA operation on matrices A, B and C which will compute $C = A - C$.

The next step of the computation is squaring the difference that was calculated above, as well as accumulating with the previously computed dimensions. Thus, we take the result from C, which contains the difference between the coordinates of the point a and the points e, f, g, h, and we fill matrices A and B, in row-major and column-major orders respectively. We then compute $D = A \times B + D$, which consists of computing $D = C \times C^t + D$. At the end of the computation, the matrix D contains the Euclidean distance between a and e, f, g, h, which is stored in the diagonal of the matrix.

To summarize, this method only computes 4 Euclidean distances, out of the 16 results that D can store. Using the WMMA API and 16×16 matrices, META-1Q is capable of computing the Euclidean distance between 1 and 16 points in 16 dimensions at a time per active warp. When using FP16 for multiplication, both FP16 and FP32 can be used in the accumulation phase.

6.3.2 META-16Q Algorithm

The second algorithm we present in this section is META-16Q. This algorithm, contrary to META-1Q, is capable of computing the Euclidean distances between 16 points and 16 other points in 16 dimensions at a time per active warp. In comparison, this method is capable of computing up to $16 \times 16 = 256$ Euclidean distances, compared to 16 using META-1Q. We illustrate the META-16Q algorithm in Figure 6.1.

META-16Q is similar to the algorithm proposed by Gallet and Gowanlock [38], but uses FP16/FP32 instead of exclusively using FP64. Hence, it also uses the expanded formula of the Euclidean distance, that we give in Equation 6.2. Furthermore, we also precompute the a^2 and b^2 parts of the formula, as they are reused many times throughout the computation. We define these as the precomputed squared coordinates and correspond to the squaring of a coordinate and its accumulation to other squared coordinates of a point, in groups of 16 (for illustrative purposes only, here in groups of 4).

Still taking two sets of four points a, b, c, d and e, f, g, h in d dimensions, using META-16Q we can compute the $4 \times 4 = 16$ Euclidean distances at the same time. Hence, we fill matrix A with 4 coordinates of the points a, b, c, d in row-major, matrix B with 4 coordinates of the points e, f, g, h in column-major, and matrix C with the precomputed squared coordinates of e, f, g, h in row-major order, that are replicated across all rows of C. We then scale matrix A by a factor -2.0 using CUDA cores, and compute the MMA operation $T = A \times B + C$. At this stage, we have computed the equivalent of $-2ab + b^2$ from Equation 6.2. Hence, we need to accumulate the precomputed squared coordinates of points a, b, c, d, as well as the Euclidean distances of previous dimensions. While we could use TCs for these operations instead of CUDA cores, it would require much more work. In particular, we would have to compute a full MMA operation instead of only computing an accumulation. Hence, we fall back to the CUDA cores for this step. At the end of the computation, we have stored in memory the $4 \times 4 = 16$ Euclidean distances we wanted to compute. Using the WMMA API, we would compute up to $16 \times 16 = 256$ Euclidean distances.

6.3.3 META-16Q and FP16

As mentioned in Section 6.3.1, META-1Q is capable of combining FP16 multiplications with either FP16 or FP32 accumulations. However, we observed that this was not the case for the META-16Q algorithm, where we are forced to accumulate using FP32 exclusively. When using FP16 only, we detected that catastrophic cancellation was occurring [26, 39], which is a numerical error that we describe as follows.

Catastrophic cancellation occurs when subtracting two numbers that are very close to each other, or when adding them using different signs (one positive and one negative), which leads to a loss of precision in the final result. The loss of precision can be further amplified when using numbers that are the product of floating-point operations, which are well-known to introduce rounding errors.

In the case of the META-16Q algorithm and using Equation 6.2, we compute $-2ab + a^2$ using the TCs, where a^2 is already precomputed. In general, we compute an addition using two different signs: $-2ab$ on the left and a^2 on the right, which is subject to catastrophic cancellation. Additionally, precomputing a^2 may have introduced rounding errors from the multiplication if not enough bits were used (e.g., if using FP16), and multiplying $-2a$ and b might also introduce rounding errors since we are forced to use FP16. Thus, this step of the computation is adding two numbers with different signs, that might already carry rounding errors. Furthermore, if the Euclidean distance is used for distance similarity searches to find points that are close to a given query point, the coordinates of those points are very likely to be numerically similar. As mentioned above, the subtraction of two close numbers is susceptible to catastrophic cancellation. Because $-2ab < a^2$, the result of this step tends to be negative. Thus, when adding the result of this operation with b^2, as outlined in Equation 6.2 we are, once more, adding two numbers with opposite signs that might carry rounding errors, and that are very likely numerically similar.

To avoid this phenomenon of catastrophic cancellation, we limit the amount of rounding errors that are introduced by accumulating using FP32 instead of FP16. Because Gallet and Gowanlock [38] are using FP64, they most likely did not observe this phenomenon and, therefore, did not mention this problem in their paper. We believe that this phenomenon is relatively obscure and yet of great importance, as it can make the result of a distance calculation completely incorrect and unusable. Note that, while we also likely subtract close numbers when using the META-1Q algorithm, these numbers are actual coordinates, have not been rounded from floating-point computation, and are of the same sign. Hence, this algorithm does not suffer from catastrophic cancellation, and it is safe to exclusively use FP16.

6.3.4 META Memory Layout

The TCs algorithm proposed by Gallet and Gowanlock [38] often suffers from very low cache hit rates, which is due to how they manage the layout of their dataset. They keep the dataset unordered and, when computing Euclidean distances for similarity searches while using an index to find the candidate points to refine, these candidate points might spread out greatly. The function to copy data from memory to a fragment requires a starting address, as well as a padding between a line of consecutive elements to copy. When using 16 × 16 matrices, this function copies 16 elements, pads the memory address, copies 16 elements, etc. When computing brute-force searches, then the candidate points can be processes iteratively in the order they are stored in memory. However, when computing similarity searches while using an index, the candidate points may not be separated by a fixed number of elements. Thus, Gallet and Gowanlock are forced to page them into shared memory, adding an extra step to their computation and also not making great use of GPU cache lines as candidate points are unlikely to be stored nearby in memory.

Similarly to TED-Join [38], when implemented into a distance similarity search algorithm, META uses a grid indexing to prune the number of candidate points to refine. Given that all the threads of the warps compute the same set of candidate points at the same time, we reorder the dataset based on the grid indexing: for each cell, all the points are grouped next to each other in memory. Hence, the padding we use to copy data from memory to the fragment is always the same, which allows us to bypass the shared-memory paging step reducing the number of registers required per thread, to benefit from greater cache hit rates, thus greatly improving performance as we will show in Section 6.4.4.

6.3.5 Using META for Distance Similarity Self-Joins

The algorithm we propose in this paper, META, is an algorithm to compute Euclidean distances, which are commonly used in data analytics applications. Thus, as mentioned before, we select the distance similarity self-join problem as a real-world case study to further evaluate the performance of META. TED-Join, proposed by Gallet and Gowanlock [38], is itself on the GDS-Join algorithm proposed by Gowanlock and Karsin [44] when computing distance similarity self-joins. Similarly, we base our version of META to compute distance similarity self-joins on GDS-Join as well. Thus, further details can be found in the papers from Gallet and Gowanlock [38] and Gowanlock and Karsin [44].

META uses a grid index to prune the number of distance calculations to compute. The grid cells are ϵ-wide and, for a given query points, finding the candidate points consists of looking for the points in the surrounding cells of the query point. Hence, within a cell, all query points have the same set of candidate points to refine. When using META-1Q, each warp is assigned one query point to compute, and will refine all the candidate points as found when searching the index. Using META-16Q, each warp is assigned up to 16 query points, all located within the same grid cell, so that they will have the same set of candidate points to refine, as found when searching the index.

Table 6.1: Exponentially distributed synthetic datasets used in the experimental evaluation.

Usage	d	n
Brute-force	16, 64, 128, 256	$2^{\{16,\dots,21\}}$
Distance similarity self-join	8, 16, 32	2M

Table 6.2: Real-world datasets used in the experimental evaluation.

Dataset	d	n	Dataset	d	n
SW3DA [69]	3	1.86M	*SuSy* [11]	18	5M
BigCross [3]	57	11M	*Songs* [18]	90	515K

6.4 Experimental Evaluation

6.4.1 Datasets

We evaluate the algorithms using a wide range of real-world and synthetic datasets, spanning several sizes, dimensionalities, and distributions. Synthetic datasets are generated following an exponential distribution, and their name is prefixed by *Expo*, followed by the dimensionality and the number of points (*Expo8D2M* is thus an exponentially distributed 8-D dataset containing 2M points). We summarize the different synthetic datasets that we use in Table 6.1, and the real-world datasets in Table 6.2. Note that some of the synthetic datasets are used only for brute-force searches, in order to evaluate the performance of CUDA and TCs when only computing Euclidean distances, while the rest of the datasets are used to compute distance similarity self-joins and use the grid index presented above.

The selectivity s represents the average number of neighboring points of each query points and found within ϵ when computing a distance similarity self-join, excluding the query points finding themselves. We calculate s as follows: $s = (|R| - |V|)/|V|$, where R is the result set (pairs of points within ϵ of each other) of the distance similarity self-join and V is the dataset. The selectivity is often used to evaluate the complexity of the search according to the value of ϵ, where increasing ϵ increases the selectivity and the amount of work (distance calculations) computed.

Table 6.3: Summary of implementation names across different levels of precision. The suffixes are as follows: "hh" refers to multiplication and accumulation in FP16, "hs" refers to multiplication in FP16 and accumulation in FP32, and "dd" refers to multiplication and accumulation in FP64.

Algorithm	Core Type	FP16	FP16-FP32	FP64
META-1Q	Tensor	META-1Q-hh	META-1Q-hs	-
META-16Q	Tensor	-	META-16Q-hs	-
TED-JOIN	Tensor	-	-	TED-JOIN-dd
GDS-JOIN	CUDA	GDS-JOIN-hh	GDS-JOIN-hs	GDS-JOIN-dd

6.4.2 Methodology

We conducted our experiments on a platform containing 2× AMD Epyc 7542 CPUs (2×32 cores, 2.9GHz), 512 GiB of RAM, which is equipped with and an Nvidia A100 GPU.

In what follows we outline our algorithm implementations and their configurations. All algorithms can compute in both brute force mode and distance similarity self-join mode. The brute force mode demonstrates the raw performance of the TC vs. CUDA core algorithms (META vs. GDS-JOIN). The distance similarity self-join demonstrates the performance when the Euclidean distance calculations are embedded in a larger application. We use the names outlined in Table 6.3 which specify the precision of the data types used in the suffix. These implementations are described as follows.

- **META:** The TCs algorithm we propose in this paper, configured with 4 warps per block, using the grid index, distances short-circuiting and dataset reordering optimizations presented above. The two variants of META are META-1Q (Section 6.3.1) and META-16Q (Section 6.3.2).

- **GDS-Join:** Parallel GPU CUDA cores distance similarity self-join algorithm proposed by Gowanlock and Karsin [44]. The algorithm is configured to use 256 threads per block, the grid index and the distance short-circuiting optimizations. For experimentation purposes, we also test using our dataset reordering technique to this algorithm. GDS-JOIN is capable of computing in either FP16 (GDS-JOIN-hh), mixed

FP16-FP32 (GDS-Join-hs) or FP64 (GDS-Join).

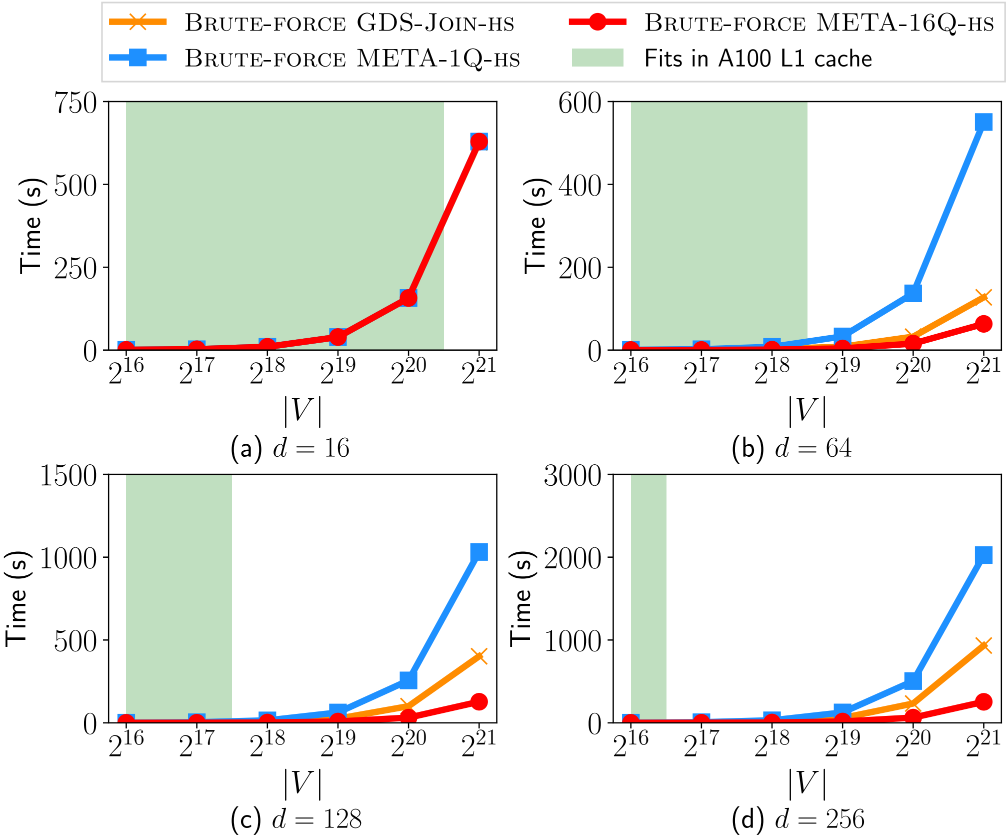
- **TED-Join:** Parallel TCs algorithm to compute Euclidean distances, incorporated into GDS-Join by Gallet and Gowanlock [38] to compute distance similarity self-joins. The algorithm uses 4 warps per block, the grid index and the distance short-circuiting optimizations from GDS-Join. We did not add the reordering dataset optimization to TED-Join. Finally, TED-Join is only capable of computing using FP64.

The algorithms are compiled using NVCC v11.2 and the O3 optimization. The reported execution times are averaged across three executions. We use the Nvidia Nsight Compute profiler to understand the performance characteristics of the algorithms and identify potential bottlenecks. Note that we do not use any CPU algorithm, as there are none capable of using FP16 data types.

6.4.3 Brute-Force Performance

We evaluate and compare the performance of only computing Euclidean distances using CUDA and TCs, i.e., using the GDS-Join and META algorithms. Focusing on Euclidean distances only allows us to address the raw performance of CUDA and TCs, as the algorithms do not perform operations other than minor host-side tasks. When computing brute-force searches, the overall complexity is $O(|V|^2)$. Because META-16Q can only use mixed FP16-FP32 precision, we evaluate META-1Q and GDS-Join using mixed precision as well.

Figure 6.2 plots the response time of META-1Q, META-16Q and GDS-Join when computing brute-force searches, on exponentially distributed synthetic datasets where $d = 16, 64, 128, 256$ and using mixed FP16-FP32 precision. We observe that the CUDA cores are consistently more efficient than META-1Q, while META-16Q is consistently outperforming GDS-Join. We explain this by the naturally higher throughput of TCs than CUDA cores, and by the fact that META-1Q does not make use of all the computational throughput capabilities of the GPU, as explained in Section 6.3.1. We profile these algorithms on



(a) $d = 16$

(b) $d = 64$

(c) $d = 128$

(d) $d = 256$

Figure 6.2: Response time of META-1Q and META-16Q versus GDS-Join, using brute-force searches to compute Euclidean distance calculations on a selection of exponentially distributed synthetic datasets and as a function of the dataset size $|V|$. The computation uses mixed FP16-FP32 precision.

the datasets going up to 2^{20} points using the Nvidia Nsight Compute profiler. The Nvidia A100 is equipped with 40 MB of L1 cache, and Figure 6.2 shows the input sizes that can fit entirely in L1 cache, as denoted by the shaded green area.

We observe that as the datasets exceed the cache capacity, the META-16Q brute-force algorithm outperforms both GDS-Join and META-1Q, and GDS-Join outperforms META-1Q. Additionally, we measure that META-1Q transfers a significantly higher volume of data than the two other algorithms: for 16 candidate points copied for the META-16Q algorithm, up to 256 Euclidean distances are computed vs. just 16 for the META-1Q algorithm, which results in more data copies.

We report profiler results in Table 6.4, collected on the 2^{20} points 64-D dataset (Fig-

Table 6.4: Profiling results of brute-force GDS-Join, META-1Q and META-16Q using mixed FP16-FP32 computation on the 2^{20} points 64-D synthetic dataset.

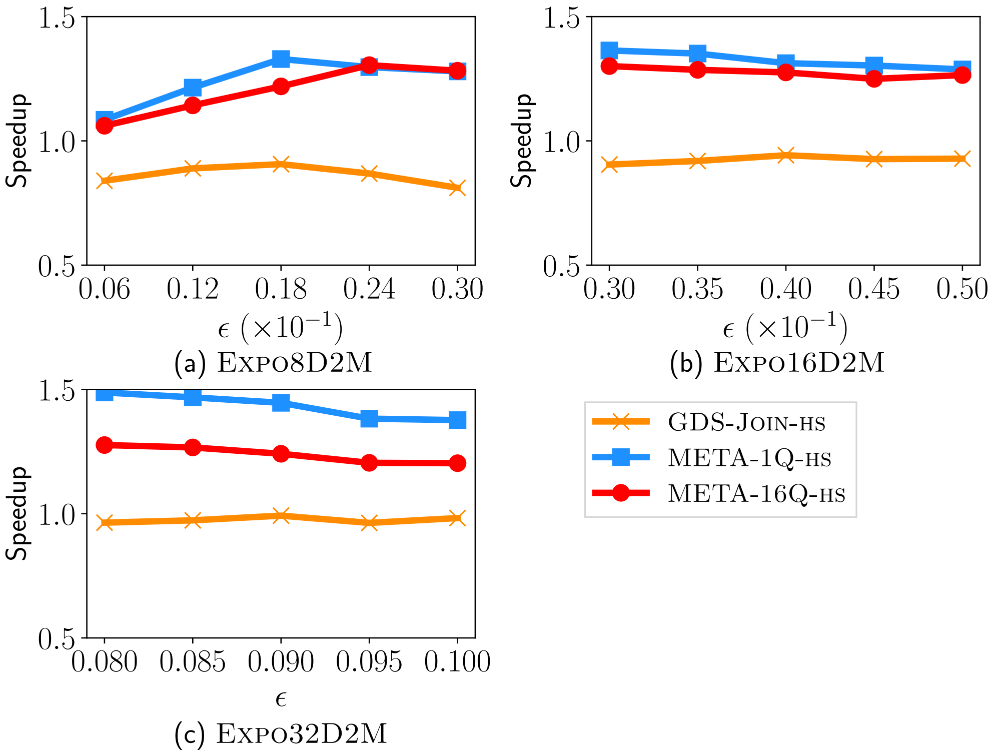
Metric	GDS-Join-hs	META-1Q-hs	META-16Q-hs
Global to L2	5.32 GB	100.82 TB	2.14 TB
L2 to L1	545.45 GB	135.43 TB	12.04 TB
L2 Hit Rate	99.60%	57.38%	83.53%
L1 Hit Rate	99.24%	52.63%	49.81%

ure 6.2(b)). These results show that META-1Q suffers from low cache performance and many accessed data. Despite GDS-Join having better metrics than META-16Q: 550.77 GB vs. 14.18 TB data accessed and 99.24% vs. 49.81% L1 cache hit rate, META-16Q performs better due to the higher computational throughput of TCs. This experiment shows that an efficient TCs algorithm yields a higher computational throughput than CUDA cores, despite having more and potentially irregular memory accesses. On average, META-1Q achieves a slowdown of 0.53× over GDS-Join, while META-16Q achieves a speedup of 5.65× and 2.32× compared to META-1Q and GDS-Join, respectively.

6.4.4 Dataset Reordering Performance

The dataset reordering optimization proposed in Section 6.3.4 aims to improve locality by storing points in the same grid cell contiguously in memory. We measure the efficiency of this optimization on META-1Q, META-16Q and GDS-Join using mixed FP16-FP32 precision.

Figure 6.3 plots the speedup of using the reordering dataset optimization (Section 6.3.4) on the META-1Q, META-16Q and GDS-Join algorithms on a selection of exponentially distributed synthetic datasets. This optimization is highly beneficial to META-1Q and META-16Q but significantly reduces the performance of GDS-Join. When using this optimization, GDS-Join yields a slowdown of between 0.86× and 0.97×, while META-1Q and META-16Q achieve a speedup between 1.24× and 1.43×, and 1.20× and 1.28×, respectively. Hence, this dataset reordering optimization always yields a slowdown for GDS-Join and a speedup for META.



(a) Expo8D2M

(b) Expo16D2M

(c) Expo32D2M

Figure 6.3: Speedup of META-1Q, META-16Q and GDS-Join using the reordering dataset optimization, and using mixed FP16-FP32 precision on a selection of exponentially distributed synthetic datasets.

We profile the three algorithms to understand the performance impacts of this optimization. For GDS-Join, we were not able to pinpoint a metric that could explain the slowdown, as profiling with and without the sorting optimization yielded very similar results. Regarding META-1Q and META-16Q, using this optimization reduces both the number of registers required per thread and the amount of shared memory that is used. Consequently, this increases the occupancy as well as the active number of warps per GPU multiprocessor, thus making better use of the resources.

6.4.5 Catastrophic Cancellation

We present here a concrete example of catastrophic cancellation we observed during our experimental evaluation. Let q be the first point of the *SuSy* dataset. We compute the distance between q and itself, which is a common operation when computing a distance

Table 6.5: Example of the catastrophic cancellation problem when computing the distance between the first point from the *SuSy* dataset (Table 6.2) and itself.

Formula	FP16	FP16-FP32	FP64
$\sqrt{\sum_{i=1}^{18}(q_i - q_i)^2}$	0.0	0.0	0.0
$\sqrt{\sum_{i=1}^{18} q_i^2 - 2q_iq_i + q_i^2}$	$\sqrt{-0.000295}$	0.0	0.0

similarity self-join for example. Mathematically, the distance between q and itself equals 0.0. We report in Table 6.5 the distance calculated using different precisions, and using the Euclidean distance formulas presented in Section 6.2.1. Note that while we show the result using CUDA cores, we would measure similar results using TCs. As mentioned above, only the expanded formula and when using only FP16 causes catastrophic cancellation to happen. Consequently, we are not able to propose an FP16-only version of our META-16Q algorithm.

6.4.6 Accuracy of FP16, FP16-FP32, and FP64

Increasing the precision of the computation also increases its cost: it is computationally more expensive to compute in FP64 than in FP32, and in FP32 than in FP16, and so on. Higher precision data types should yield a more accurate result which, under some circumstances, can be more important than execution time alone. We thus compare the performance of using FP16 only against using mixed FP16-FP32. Note that we only test META-1Q and GDS-Join, as META-16Q can only compute using mixed precision.

Figure 6.4 plots the speedup of using FP16 over mixed FP16-FP32 precision, using the META-1Q and GDS-Join algorithms on a selection of datasets (Tables 6.1 and 6.2). As one could expect, using FP16 typically yields better performance compared to using mixed FP16-FP32 precision. However, the speedups are relatively moderate: between 1.00× and 1.18× for GDS-Join, and between 0.96× and 1.18× for META-1Q.

We report in Table 6.6 the accuracy ratio of GDS-Join, META-1Q and META-16Q across several precisions, as compared to GDS-Join using FP64, which is the ground truth

Table 6.6: Accuracy ratio of the GDS-Join, META-1Q and META-16Q algorithms as compared to the result of GDS-Join using FP64 when computing the distance similarity self-join. The percentage $p = (1 - (||R_{ref}| - |R_{comp}||)/|R_{ref}|) * 100$ where R_{ref} is the result set (the set of result pairs) returned by GDS-Join using FP64 and R_{comp} is the result set returned by the comparison algorithm.

	R_{comp} FP16		
Dataset	GDS-Join-hh	META-1Q-hh	
SW3DA	99.85%	99.86%	
SuSy	99.66%	99.92%	
BigCross	99.88%	99.79%	
Songs	96.86%	95.04%	
Expo8D2M	99.80%	99.82%	
Expo16D2M	99.54%	99.54%	
Expo32D2M	99.19%	99.23%	

	R_{comp} FP16-FP32		
Dataset	GDS-Join-hs	META-1Q-hs	META-16Q-hs
SW3DA	99.93%	99.94%	99.64%
SuSy	99.60%	99.71%	99.70%
BigCross	98.17%	99.97%	95.68%
Songs	97.02%	95.35%	98.32%
Expo8D2M	99.93%	100.0%	99.94%
Expo16D2M	99.88%	100.0%	99.87%
Expo32D2M	99.71%	99.95%	99.68%

	R_{comp} FP64		
Dataset	GDS-Join-dd		TED-Join-dd
SW3DA	100.0%		99.97%
SuSy	100.0%		100.0%
BigCross	100.0%		100.0%
Songs	100.0%		100.0%
Expo8D2M	100.0%		100.0%
Expo16D2M	100.0%		100.0%
Expo32D2M	100.0%		100.0%

implementation. We find that, overall, using a low precision such as FP16 to compute Euclidean distances yields a very high accuracy compared to mixed FP16-FP32 and even FP64. For example, on FP16, the lowest accuracy is on *Songs*, where we obtain a 95.04% accuracy compared to using GDS-Join with FP64. In 6 of 7 datasets (excluding *Songs*), we obtain over 99% accuracy with FP16. Interestingly, comparing FP16 to FP16-FP32 for META-1Q the accuracy improves on 6 of 7 datasets (excluding *SuSy*). Furthermore, META-1Q typically achieves a better accuracy compared to GDS-Join FP64 than GDS-Join-hh or GDS-Join-hs. On the TC algorithms, we find that the accuracy improves as we increase the precision of the data types. Therefore, using a lower precision datatype such as FP16 or FP16-FP32 is dependent on the application. If the application can tolerate some loss of accuracy, it may be worthwhile to trade accuracy for performance, since low precision data types are more efficient (Figure 6.4).

6.4.7 Algorithm Performance Comparisons

We compare in this section the performance of the three algorithms: META-1Q, META-16Q and GDS-Join. Because the dataset reordering optimization is only beneficial to META-1Q and META-16Q, we do not use this optimization for GDS-Join. For comparison purposes, all three algorithms use mixed FP16-FP32 precision.

We show in Figure 6.5 the execution time of GDS-Join, META-1Q and META-16Q using mixed FP16-FP32 and GDS-Join and TED-Join using FP64 on a selection of datasets (Tables 6.1 and 6.2). GDS-Join and TED-Join do not use the reordering dataset optimization, while META-1Q and META-16Q do. We see that META-1Q is able to perform better than GDS-Join, particularly when ϵ increases and the total number of distance calculations as well, and outperforms META-16Q as well when in lower dimensions (Figure 6.5(a) and (e)). Regarding META-16Q, it consistently outperforms GDS-Join in all scenarios. We report in Table 6.7 the minimum, maximum and average speedup of META-1Q and META-16Q over GDS-Join using mixed FP16-FP32. On average, both META-1Q and META-16Q outperform GDS-Join. Finally, and as we could expect, our mixed FP16-FP32 tensor algorithms typically perform better than FP64 algorithms, as the workload of lower precisions (here mixed FP16-FP32) is lower than when using higher precisions (here FP64).

Using the Nvidia Nsight Compute profiler, we observe that the major difference between

Table 6.7: Minimum, maximum and average speedup of META-1Q and META-16Q vs. GDS-Join using mixed FP16-FP32 precision, as well as GDS-Join and TED-Join using FP64, on the same selection of datasets as shown in Figure 6.5.

	GDS-Join-hs	
Speedup	META-1Q-hs	META-16Q-hs
Min	0.30×	0.66×
Max	5.38×	2.74×
Avg	1.38×	1.46×

	GDS-Join-dd	
Speedup	META-1Q-hs	META-16Q-hs
Min	0.43×	0.77×
Max	6.45×	4.86×
Avg	1.66×	1.80×

	TED-Join-dd	
Speedup	META-1Q-hs	META-16Q-hs
Min	0.59×	0.80×
Max	2.94×	3.92×
Avg	1.20×	1.52×

META-1Q and META-16Q is that, in lower dimensions, META-16Q is limited by its higher register usage, which is compensated in higher dimensions by the higher number of Euclidean distances to compute and its ability to compute up to 16× more distances at a time than META-1Q.

6.5 Discussion and Conclusion

This paper presents the first low precision Euclidean distance algorithm for TCs, which are used as a building block for many algorithms, such as nearest neighbor searches or clustering. We propose two variants of META, including META-1Q, which computes the Euclidean distance between 1 point and 16 other points in either FP16 or mixed FP16-FP32; and, META-16Q, which computes the Euclidean distance between up to 16 points and 16 other points, but only in mixed FP16-FP32 precision. We illustrated the performance of the algorithm in the brute-force context, where many distance calculations need to be computed, which is often the case when computing forces between bodies in a scientific simulation [28]. We also presented a case study application for META by incorporating it into an existing state-of-the-art distance similarity self-join algorithm for the GPU that uses CUDA cores (GDS-Join).

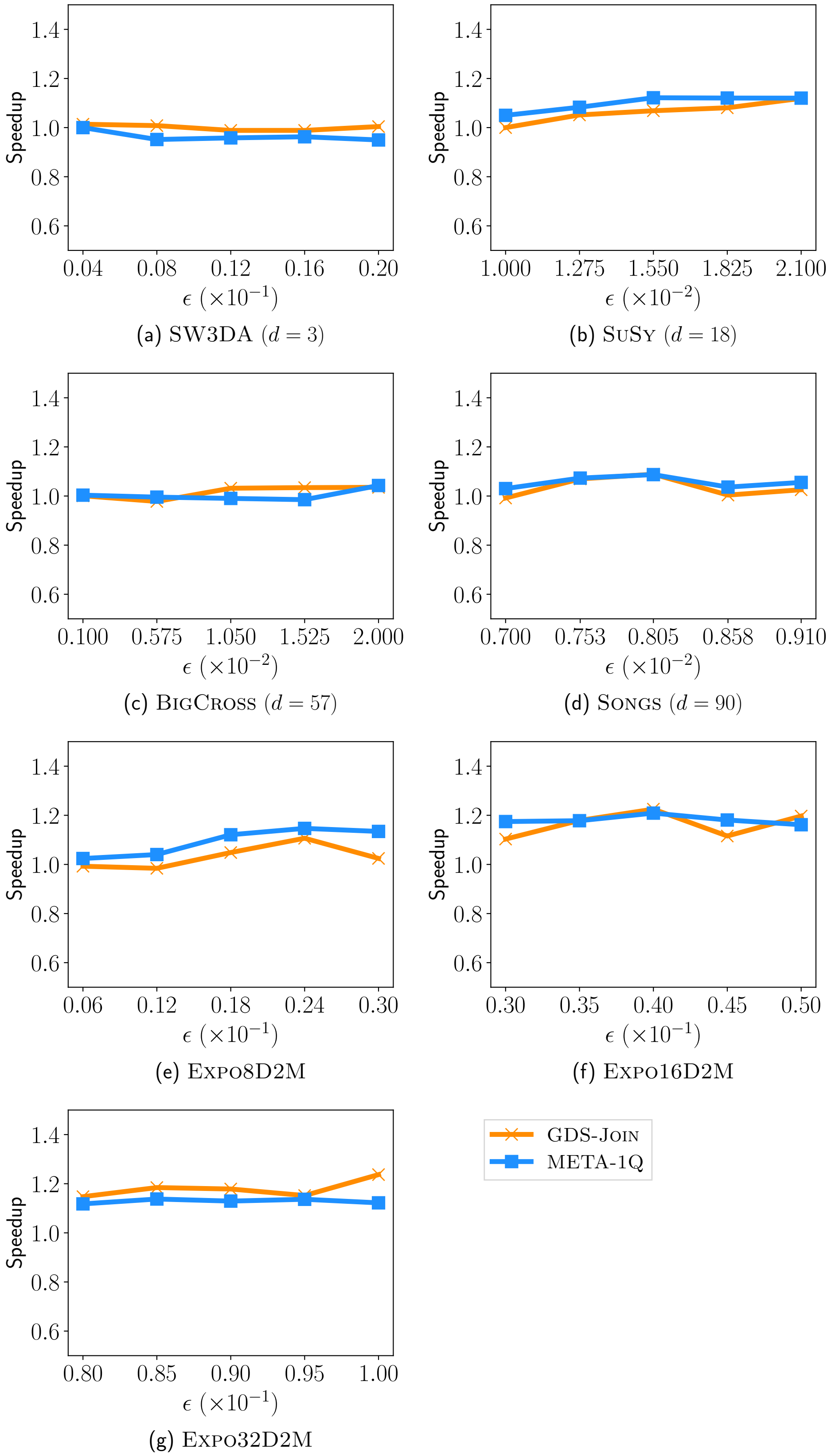
We evaluated META-1Q and META-16Q against GDS-Join. Across a broad range of datasets spanning several dimensionalities, sizes and distributions, we observe that META-16Q consistently outperforms GDS-Join (the CUDA core algorithm), whether the algorithms use brute-force searches (Figure 6.2) or an index (Figure 6.5). META-1Q is usually outperformed by both META-16Q and GDS-Join, except in lower dimensions ($d \leq 8$), where it is the most efficient algorithm. In summary, META is generally more efficient than GDS-Join: the META-1Q variant should be used when $d \leq 8$, and META-16Q should be used otherwise.

We evaluated using FP16 instead of mixed FP16-FP32, both in terms of performance and accuracy. We find that using FP16 only has a marginal impact on the accuracy of the computation compared to FP16-FP32, while using FP16 is only marginally faster than FP16-FP32. Consequently, it is not clear whether FP16 or FP16-FP32 should be used in the case where an algorithm can tolerate minimal losses to accuracy.

Future plans include implementing META into other data-analysis applications, such as nearest neighbor searches or clustering. Furthermore, because TCs and CUDA cores are physically distinct, we believe that concurrently leveraging these two processors to compute Euclidean distances should further improve performance, but this remains to be investigated. Our literature review showed that using TCs for increasingly general-purpose computations is still relatively rare; therefore, we believe that there are many research avenues for the community to investigate in this area.

Acknowledgements

This material is based upon work supported by the National Science Foundation under Grant No. 2042155.



(a) SW3DA ($d = 3$)

(b) SuSy ($d = 18$)

(c) BigCross ($d = 57$)

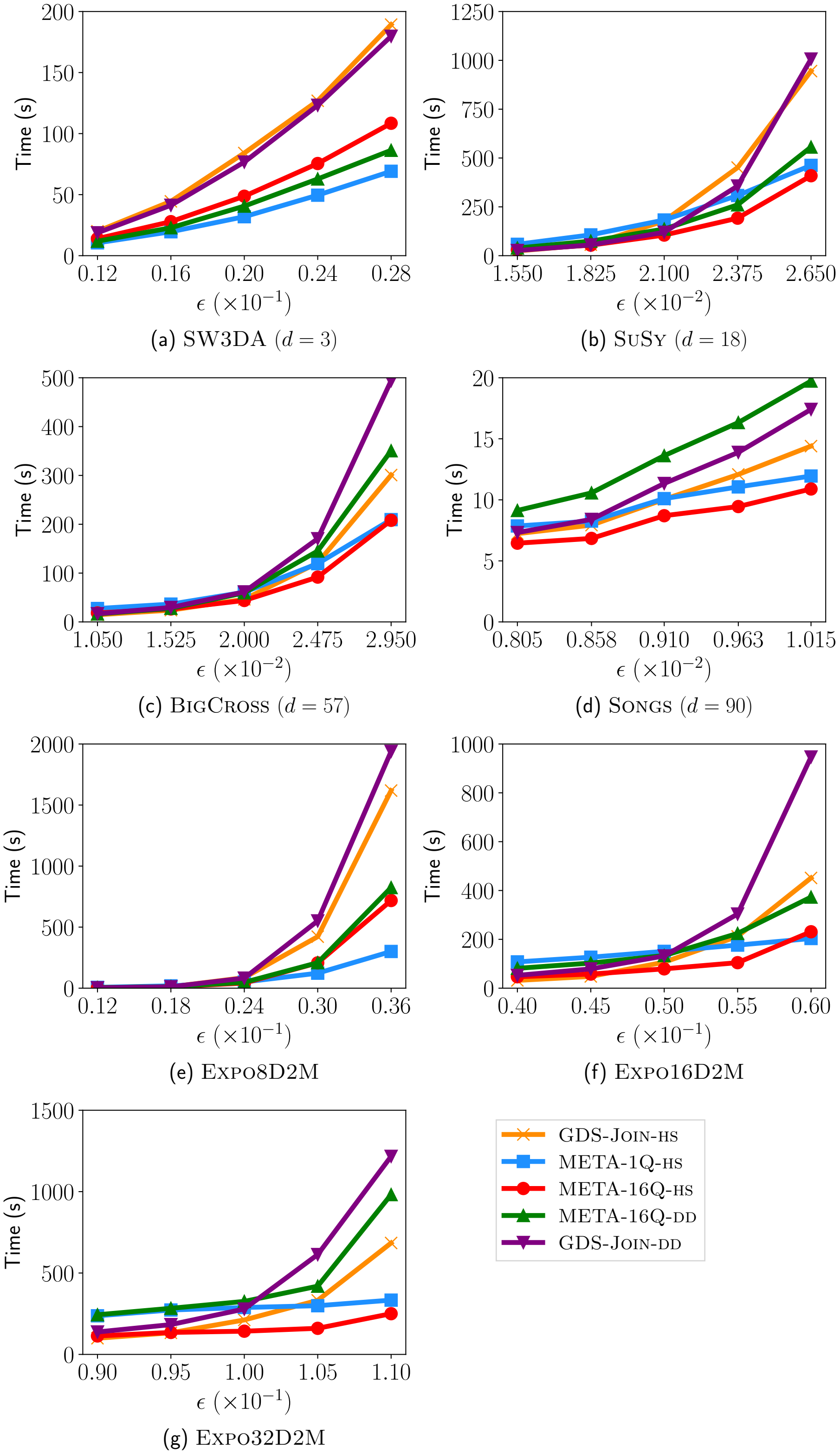
(d) Songs ($d = 90$)

(e) Expo8D2M

(f) Expo16D2M

(g) Expo32D2M

Figure 6.4: Execution time ratio of META-1Q and GDS-Join using FP16 over mixed FP16-FP32 on a selection of datasets presented in Tables 6.1 and 6.2.



(a) SW3DA ($d = 3$)

(b) SuSy ($d = 18$)

(c) BigCross ($d = 57$)

(d) Songs ($d = 90$)

(e) Expo8D2M

(f) Expo16D2M

(g) Expo32D2M

Figure 6.5: Execution time of GDS-Join, META-1Q and META-16Q using mixed FP16-FP32, and GDS-Join and TED-Join using FP64, on a selection of datasets (Table 6.1 and 6.2. META-1Q and META-16Q are using the dataset reordering optimization.

Chapter 7

Discussion & Conclusion

Computing distance similarity searches, and more generally Euclidean distances, is a significant performance bottleneck in many algorithms. Thus, optimizing the performance of these relatively basic yet important operations that are building blocks for other algorithms is essential to improve the performance of these algorithms. Furthermore, computing platforms evolved to be increasingly heterogeneous, particularly with the addition of accelerators such as GPUs or ASICs within GPUs. Hence, optimizations for distance similarity searches and Euclidean distance calculations must carefully consider the architectural constraints of target platforms. We showed throughout this dissertation that this can be challenging. We summarize these challenges below.

- CPUs and GPUs have very different architectural characteristics. Consequently, an algorithm developed and optimized for the CPU is unlikely to work as well on the GPU, and vice versa. Hence, as a single algorithm should not be used on these different architectures, it is necessary to two develop two distinct versions, and the associated optimizations should be suited to each architecture.
- GPUs typically have a significantly higher peak computational throughput than CPUs. As such, it appears more interesting, in terms of performance, to develop a parallel algorithm for the GPU than it is for the CPU. However, designing a GPU algorithm leaves the CPU underutilized. When using both the CPU and GPU, we need to

determine to which architecture each part of the computation should be assigned to, based on the characteristics of the architectures. Particularly, if we consider that the GPU has a higher computational throughput than the CPU, then it should be assigned more work. However, in the case of algorithms with irregular workloads, such as computing multiple distance similarity searches where the different query points have varying workloads, then it can be challenging to (*i*) estimate the workload to compute, and (*ii*) assign it to the processor that is likely to yield a balanced workload.

- GPU architectures have significantly improved in recent years, particularly with the introduction of specific-purpose cores, in addition to the general-purpose CUDA cores already present. Leveraging these specific-purpose cores, such as TCs, may greatly improve performance of the algorithm. However, such algorithms may require substantial modifications to adapt to these very specific architectures. And, because of their recent introduction to GPUs, it can be challenging to find reliable information regarding their operation. Furthermore, many proposed work in the literature that were once state-of-the-art algorithms have the potential to be further improved now by using such novel cores.

7.1 Summary of Contributions

The work we propose in this dissertation are detailed in Chapters 3 to 6, and that we summarize in Table 7.1 and the sections below:

7.1.1 Load Imbalance Mitigation Optimizations for GPU-Accelerated Similarity Joins

Chapter 3 consists of a peer-reviewed publication [35], where we proposed several optimizations to an existing state-of-the-art GPU distance similarity search algorithm. *Note that while this publication makes several contributions, we only summarize here the contributions*

Table 7.1: Summary of the contributions of this dissertation, sorted by processor (CPU/GPU) and by examined algorithm: Distance Similarity Searches (DSS) or Euclidean Distances (ED).

Chapter	CPU	CUDA	TCs	DSS	ED	Main contributions
Chapter 3		✓		✓		Intra- and inter-warp workload balancing.
Chapter 4	✓	✓		✓		Heterogeneous CPU-GPU algorithm, both processors computing DSS.
Chapter 5			✓		✓	Using TCs to compute FP64 ED.
Chapter 6			✓		✓	Using TCs to compute FP16 and mixed FP16-FP32 ED.

that were conducted during the completion of this dissertation.

The most significant contribution of this chapter is the improved resource utilization of the distance similarity search GPU algorithm. As previously stated, when computing distance similarity searches, the query points are extremely likely to have different workloads. By default, the GPU algorithm assigns a query point to a GPU thread, regardless of this particularity. Hence, as GPU threads are grouped in warps (32 threads) starting and ending their execution at the same time, some threads were executing for a longer period of time than other threads of the warp, because they were assigned query points with a higher workload to compute. By using the grid indexing method already implemented, we were able to efficiently calculate the workload of the query points with very minimal overhead. With this information, we then designed a queue mechanism, where query points are sorted by their workload, and where the threads of a warp are assigned contiguous points from the queue. Thus, the query points assigned to a warp all have a very similar workload and thus have minimal variance in their workload. Consequently, the threads of a warp are more likely to end their work at the same time, improving their overall utilization (as measured with the warp execution efficiency metric discussed in this chapter), and incidentally the performance as well.

This solution to the workload imbalance problem was revealed to have very minimal overhead, while greatly improving performance. As such, this optimization was used in every subsequent algorithm we designed and presented in this dissertation. More particularly, this solution to the workload balancing problem was key to the heterogeneous CPU-GPU algorithm we proposed in Chapter 4 and that we summarize in the following section.

7.1.2 Heterogeneous CPU-GPU Epsilon Grid Joins: Static and Dynamic Work Partitioning Strategies

Chapter 4 consists of two peer-reviewed publications [36, 37], where we proposed a heterogeneous CPU-GPU distance similarity search algorithm, and where we explored several methods to partition the work between the processors, while leveraging the workload balancing mechanism developed in Chapter 3.

We explained throughout this dissertation the main differences between CPUs and GPUs, describing the importance of using both when possible, while also highlighting the issues related to leveraging heterogeneous architectures. Among the issues, having the best algorithm for each architecture is essential to get the best performance possible. Hence, a single and optimized algorithm can not be used, due to it being specific to a single architecture. Consequently, we proposed to combine two existing algorithms: the state-of-the-art parallel CPU algorithm Super-EGO [57], and the GPU algorithm we proposed in Chapter 3. By doing this, we were able to get the best performance possible on the CPU and GPU. We slightly modified Super-EGO to be consistent with our GPU algorithm. We then reworked both algorithms using the queue mechanism that was originally designed for the GPU algorithm, which we migrated to the host so the CPU algorithm could use it as well. Using this queue, both the CPU and GPU are assigned work on-demand. Remember that the query points in the queue are sorted by their workload. Hence, because the GPU is typically significantly more efficient than the CPU, the algorithm starts by assigning the most computationally expensive query points (i.e., the ones at the start of the queue) to the GPU, and the least computationally expensive (i.e., those at the end of the queue) to the CPU.

The other main issue of the heterogeneous CPU-GPU algorithm for distance similarity searches is the different computational throughput of both architectures and the irregular workload of the problem itself. Hence, we explored several workload partitioning solutions in an attempt to find the most efficient one. We tried two main methods: dynamic and static work partitioning. The dynamic partitioning method consists of having both the CPU and GPU request query points to compute from the queue until it is empty. The challenge here is to assign chunks that are neither too big to ensure a relative workload balance between the processors, but also large enough to saturate GPU resources in particular. The second method we explored is static partitioning, where the workload of each processor is first determined at runtime, based on their proportional estimated computational throughput. Using this method, we also explored two approaches: partitioning based on the total number of query points but making no assumptions on their actual workloads, and partitioning based on the workload of the query points. Thus, if we consider that the GPU corresponds to 80% of the performance of a platform and the CPU 20%, in the first case we would attribute 80% of the query points to the GPU, and 20% to the CPU. In the second case, we would attribute 80% of the total workload to the GPU, which could correspond to 20% of the query points for example, depending on their respective workload (i.e., 20% of the query points account for 80% of the total work to compute, while 20% of the total work to compute might account for 80% of the query points). Across a broad range of experiments, we observed that the dynamic partitioning method is typically the most efficient, as it does not need to make assumptions about the throughput of the architectures. We then find that the static partitioning based on the query points performs the worst on average, as it can not accurately estimate the entire workload to compute.

This heterogeneous CPU-GPU distance similarity search algorithm leveraged two existing state-of-the-art algorithms, including one we proposed in Chapter 3, and explored different work partitioning strategies. Overall, this algorithm was able to alleviate the most important issues of heterogeneous CPU-GPU algorithms: have an efficient algorithm for each architecture, and efficiently partition the work, while making better overall usage of compute resources than other existing algorithms.

7.1.3 Leveraging GPU Tensor Cores for Double Precision Euclidean Distance Calculations

Chapter 5 consists of a peer-reviewed publication [38], where we propose to leverage TCs, recently added to GPUs, to compute Euclidean distances and that we apply to compute distance similarity searches as well.

TCs are specific-purpose cores that are equipped on several recent generations of GPUs and are designed to compute MMA operations, with significantly higher throughput than if the operation was computed using general-purpose (CUDA) cores. Despite their high specificity and that they are heavily marketed for machine learning and related fields, any computation that can be expressed as MMA operations can leverage TCs, which may yield improved performance. As such, we investigated using TCs to compute Euclidean distances in lieu of the general-purpose CUDA cores as we did in the previously proposed GPU algorithms.

Using the WMMA API for FP64 computations, we used 8×4 and 4×8 matrices for the product part of the MMA, and 8×8 matrices for the accumulation (as restricted by the API). As we explained in Chapter 5, we used the expanded form of the Euclidean distance (Equation 2.2) to alleviate the restrictions of the matrix sizes and the usual Euclidean distance formula (Equation 2.1). Because the formula we use is not a perfect balance of additions and multiplications, we optimize the overall computation by pre-computing the squared terms in the formula (a^2 and b^2). Furthermore, because the number of accumulations is greater than the number of multiplications, we use CUDA cores to perform those additional accumulations rather than TCs, where we would have to perform unnecessary multiplications as well (as part of an MMA operation). As such, our algorithm can compute, per warp, the Euclidean distance between up to 8 query points and 8 candidate points, 4 dimensions at a time, achieving a good performance improvement in lower dimensional cases (typically $d \leq 8$), with a precision similar to using CUDA cores.

7.1.4 Optimizing Euclidean Distance Calculations in Low Precision with GPU Tensor Cores

Chapter 6 consists of a manuscript currently under review at the time of writing, where we propose to build upon the previous publication [38] presented in Chapter 5 and summarized in Section 7.1.3, to propose several TC algorithms to compute Euclidean distances but using a lower precision than before (FP16 and mixed FP16-FP32 instead of FP64).

The TCs algorithm presented in Chapter 5 is only capable of computing Euclidean distances using FP64, which is not available on every Nvidia GPU possessing TCs, contrary to FP16 and mixed FP16-FP32 precision that are found on a wider range of devices. Using the previous algorithm as a starting point, this low precision algorithm uses 16×16 matrices, which allows us to use both the common and the expanded Euclidean distance formula (Equations 2.1 and 2.2). The expanded formula allows a single warp to compute the Euclidean distance between up to 16 query points and 16 candidate points, in 16 dimensions at a time. Using the regular formula, the number of query points is reduced to only 1 (instead of 16). As we would expect, the 16 query point version typically performs better than the 1 query point version. Furthermore, we measured that using lower precision TCs still obtains high precision compared to FP64, while also achieving better performance.

7.2 Future Perspectives

As we show in Section 7.1 and in Table 7.1, we identify several research paths worth exploring and that we present below.

7.2.1 Further Tensor Core Optimizations

The TC algorithms we propose and that use low precisions use 16×16 matrices. However, the WMMA API provides several other matrix sizes to use with FP16 and mixed FP16-FP32, which should be explored to determine if changing the matrix size impacts performance.

7.2.2 Heterogeneous CUDA-Tensor Euclidean Distance Calculations

In a GPU, CUDA and TCs are physically distinct. Because there are significantly more CUDA cores than TCs, an algorithm that essentially uses TCs is unlikely to make efficient and constant use of CUDA cores. Consequently, a compelling future work direction is a heterogeneous GPU-GPU algorithm that would concurrently use both CUDA and TCs to compute Euclidean distances.

Several options should be explored to determine the most efficient method, including:

- Use independent CUDA and TC kernels and use an appropriate number of threads for each, expecting both kernels to execute concurrently on a single GPU. However, because it can be challenging to predict the actual kernel execution order, we may not be able to achieve kernel concurrency, if the hardware scheduler considers that the kernels can not execute at the same time.

- Use a single kernel, and split the total number of blocks in two, with blocks that will be using CUDA cores and blocks that will use TCs. Because several blocks can be executed concurrently, using a single kernel forces this concurrency.

- Use a single kernel, and split the total number of threads of each block in two, with threads that will be using CUDA cores and threads that will use TCs. This solution should be particularly interesting if, for example, we find that the threads of each algorithm should communicate with the threads of the other algorithm, thus using shared memory.

7.2.3 Performance Modeling of Euclidean Distance Calculations for CUDA and Tensor Cores

Similarly to the modeling we did for the heterogeneous CPU-GPU algorithm to split the work between the two architectures, we would here model the performance of using CUDA and TCs to compute Euclidean distances. This modeling could then be used to (*i*) better understand the performance intrinsics of both types of cores, and (*ii*) better partition the work between CUDA and TCs. Using the future work presented in Section 7.2.2, while the simple solution would be to equivalently divide the work in two and assign equivalent workloads to the CUDA and TCs, modeling the performance of these cores would allow us a finer tuning, and allocate a different number of threads/blocks, with the expectation of yielding better performance.

7.2.4 Exploring Novel Architecture Features

GPU Ray Tracing Cores for Index Searches: Since the addition of TCs to certain GPUs, another type of specific-purpose core was added to some GPUs: the Ray-tracing (RT) core. These cores are designed to greatly improve the performance of RT algorithms and can execute two tasks. Using a Bounding Volume Hierarchy (BVH) structure on the objects of the scene, RT cores are used to first search the BVH structure, and then determine if a ray intersects any volume.

Using a similar BVH structure to index multidimensional points, we could use the RT cores to search this structure, cast a ray, and determine the adjacent volumes and the points they contain for which we should compute Euclidean distances. From there, we would leverage one of our Euclidean distance calculation algorithms we proposed, or that we plan to propose in the future including the CUDA-Tensor algorithm.

CPU Matrix Extensions for Euclidean Distance Calculations: Similarly to the addition of TCs on recent GPUs, CPUs are also starting to get equipped with accelerators specifically made for matrix operations. Similarly to the intrinsic functions [55] one can use to leverage the vectorization capabilities of most CPUs (i.e., to work on vectors), Intel is proposing a new set of extensions: the Advanced Matrix Extensions (AMX) [54]. While this new functionality is targeted for machine learning applications, similarly to the GPU TCs, we believe this novel option could be leveraged for other general-purpose applications, similarly to our GPU TC research (Chapters 5 and 6). By designing a new and efficient CPU algorithm working on matrices such as this one, we could then propose another heterogeneous CPU-GPU algorithm, similar to the one presented in Chapter 4.

Bibliography

[1] OpenStreetMap Bulk GPS Point Data, 2012. URL `https://blog.openstreetmap.org/2012/04/01/bulk-gps-point-data/`. Accessed: March 29, 2023.

[2] Gaia DR 2, 2018. URL `https://cosmos.esa.int/web/gaia/dr2`. Accessed: March 29, 2023.

[3] Marcel R. Ackermann, Marcus Märtens, Christoph Raupach, Kamil Swierkot, Christiane Lammersen, and Christian Sohler. StreamKM++: A Clustering Algorithm for Data Streams. *ACM Journal of Experimental Algorithmics*, 17, May 2012. doi: 10.1145/2133803.2184450.

[4] Thomas Ahle and Francesco Silvestri. Similarity Search with Tensor Core Units. *Similarity Search and Applications*, pages 76–84, 2020. doi: 10.1007/978-3-030-60936-8_6.

[5] S et al. Alam. The Eleventh and Twelfth Data Releases of the Sloan Digital Sky Survey: Final Data from SDSS-III. *The Astrophysical Journal Supplement Series*, 219:12, 2015.

[6] A. Andoni and P. Indyk. Near-Optimal Hashing Algorithms for Approximate Nearest Neighbor in High Dimensions. *47th Annual IEEE Symposium on Foundations of Computer Science*, pages 459–468, 2006. ISSN 0272-5428. doi: 10.1109/FOCS.2006.49.

[7] Alexandr Andoni, Piotr Indyk, Thijs Laarhoven, Ilya Razenshteyn, and Ludwig Schmidt. Practical and Optimal LSH for Angular Distance. *Advances in Neural Information Processing Systems 28*, pages 1225–1233, 2015.

[8] J Appleyard and S Yokim. Programming Tensor Cores in CUDA 9. `https://developer.nvidia.com/blog/programming-tensor-cores-cuda-9/`, 2017. Accessed: March 29, 2023.

[9] Muhammad A. Awad, Saman Ashkiani, Rob Johnson, Martín Farach-Colton, and John D. Owens. Engineering a High-performance GPU B-Tree. *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming*, pages 145–157, 2019. doi: 10.1145/3293883.3295706.

[10] Peter Bakkum and Kevin Skadron. Accelerating SQL Database Operations on a GPU with CUDA. *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, pages 94–103, 2010. doi: 10.1145/1735688.1735706.

[11] P Baldi, P Sadowski, and D Whiteson. Searching for exotic particles in high-energy physics with deep learning. *Nature Communications*, 5, 2014. doi: 10.1038/ncomms5308.

[12] R. Baraglia, G. De Francisci Morales, and C. Lucchese. Document Similarity Self-Join with MapReduce. *2010 IEEE International Conference on Data Mining*, pages 731–736, 2010. doi: 10.1109/ICDM.2010.70.

[13] R. Bayer and E. M. McCreight. Organization and Maintenance of Large Ordered Indexes. *Acta Informatica*, 1(3):173–189, 1972. ISSN 1432-0525. doi: 10.1007/BF00288683.

[14] Norbert Beckmann, Hans-Peter Kriegel, Ralf Schneider, and Bernhard Seeger. The R*-Tree: An Efficient and Robust Access Method for Points and Rectangles. *Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data*, 19:322–331, 1990. doi: 10.1145/93597.98741.

[15] R.E. Bellman. Princeton University Press, 1961. ISBN 9780691652214.

[16] Jon Louis Bentley. Multidimensional Binary Search Trees Used for Associative Searching. *Communications of the ACM*, 18(9):509–517, 1975. doi: 10.1145/361002.361007.

[17] Stefan Berchtold, Daniel A. Keim, and Hans-Peter Kriegel. The X-tree : An Index Structure for High-Dimensional Data. *Proceedings of the 22th International Conference on Very Large Data Bases*, pages 28–39, 1996.

[18] Thierry Bertin-Mahieux, Daniel P.W. Ellis, Brian Whitman, and Paul Lamere. The million song dataset. *Proceedings of the 12th International Conference on Music Information Retrieval*, 2011.

[19] Christian Böhm and Florian Krebs. The k-Nearest Neighbour Join: Turbo Charging the KDD Process. *Knowledge and Information Systems*, 6(6):728–749, 2004. ISSN 0219-3116. doi: 10.1007/s10115-003-0122-9.

[20] Christian Böhm, Bernhard Braunmüller, Markus M. Breunig, and Hans-Peter Kriegel. High Performance Clustering Based on the Similarity Join. *Proceedings of the International Conference on Information and Knowledge Management*, pages 298–305, 2000. doi: 10.1145/354756.354832.

[21] Christian Böhm, Bernhard Braunmüller, Florian Krebs, and Hans-Peter Kriegel. Epsilon Grid Order: An Algorithm for the Similarity Join on Massive High-dimensional Data. *ACM SIGMOD Record*, 30(2):379–388, 2001. doi: 10.1145/376284.375714.

[22] Christian Böhm, Robert Noll, Claudia Plant, and Andrew Zherdin. Index-supported Similarity Join on Graphics Processors. *Datenbanksysteme für Business, Technologie und Web*, pages 57–66, 2009. URL `https://dl.gi.de/20.500.12116/20493`.

[23] Martin Burtscher, Rupesh Nasre, and Keshav Pingali. A Quantitative Study of Irregular Programs on GPUs. pages 141–151, 2012. doi: 10.1109/IISWC.2012.6402918.

[24] S Chaudhuri, V Ganti, and R Kaushik. A Primitive Operator for Similarity Joins in Data Cleaning. *22nd International Conference on Data Engineering*, 2006. ISSN 1063-6382. doi: 10.1109/ICDE.2006.9.

[25] Douglas Comer. The Ubiquitous B-Tree. *ACM Computer Survey*, 11(2):121–137, 1979. ISSN 0360-0300. doi: 10.1145/356770.356776.

[26] Annie Cuyt, Brigitte Verdonk, S Becuwe, and Peter Kuterna. A Remarkable Example of Catastrophic Cancellation Unraveled. *Computing*, 66(3):309–320, May 2001. doi: 10.1007/s006070170028.

[27] Abdul Dakkak, Cheng Li, Jinjun Xiong, Isaac Gelado, and Wen-mei Hwu. Accelerating Reduction and Scan Using Tensor Core Units. *Proceedings of the ACM International Conference on Supercomputing*, page 46–57, 2019. doi: 10.1145/3330345.3331057.

[28] Vinicius Carius de Souza, Leonardo Goliatt, and Priscila V. Z. Capriles Goliatt. Clustering algorithms applied on analysis of protein molecular dynamics. *2017 IEEE Latin American Conference on Computational Intelligence (LA-CCI)*, pages 1–6, 2017. doi: 10.1109/LA-CCI.2017.8285695.

[29] Brian Donnelly and Michael Gowanlock. A Coordinate-Oblivious Index for High-Dimensional Distance Similarity Searches on the GPU. 2020. doi: 10.1145/3392717.3392768.

[30] Osman Durmaz and Hasan Sakir Bilge. Fast image similarity search by distributed locality sensitive hashing. *Pattern Recognition Letters*, 128:361–369, 2019. doi: https://doi.org/10.1016/j.patrec.2019.09.025.

[31] Ramez Elmasri and Shamkant B. Navathe. *Fundamentals of Database Systems*, chapter 18.4.1. Pearson, 7 edition, 2016.

[32] M. Ester, H. Kriegel, , J. Sander, and X Xu. A density-based algorithm for discovering clusters in large spatial databases with noise. *Proceedings of the 2nd KDD*, pages 226 – 231, 1996.

[33] R. A. Finkel and J. L. Bentley. Quad Trees: A Data Structure for Retrieval on Composite Keys. *Acta Informatica*, 4(1):1–9, 1974. ISSN 1432-0525. doi: 10.1007/BF00288933.

[34] Benoit Gallet. Gpu kernel performance optimizations for efficient similarity joins. *Master's Degree Thesis, Université d'Orléans, Northern Arizona University*, 2018.

[35] Benoit Gallet and Michael Gowanlock. Load Imbalance Mitigation Optimizations for GPU-Accelerated Similarity Joins. *Proceedings of the 2019 IEEE International Parallel and Distributed Processing Symposium Workshops*, pages 396–405, 2019. doi: 10.1109/IPDPSW.2019.00078. URL `https://github.com/benoitgallet/self-join-hpbdc2019`. Accessed: March 29, 2023.

[36] Benoit Gallet and Michael Gowanlock. HEGJoin: Heterogeneous CPU-GPU Epsilon Grids for Accelerated Distance Similarity Join. *Proceedings of the 25th International Conference on Database Systems for Advanced Applications (DASFAA)*, pages 372–388, 2020. doi: 10.1007/978-3-030-59419-0_23.

[37] Benoit Gallet and Michael Gowanlock. Heterogeneous CPU-GPU Epsilon Grid Joins: Static and Dynamic Work Partitioning Strategies. *Data Science and Engineering*, 6:39–62, 2020. doi: 10.1007/s41019-020-00145-x. URL `https://github.com/benoitgallet/HEGJoin`. Accessed: March 29, 2023.

[38] Benoit Gallet and Michael Gowanlock. Leveraging GPU Tensor Cores for Double Precision Euclidean Distance Calculations. *Proceedings of the 29th IEEE International Conference on High Performance Computing, Data, and Analytics (HiPC)*, pages 135–144, 2022. doi: 10.1109/HiPC56025.2022.00029. URL `https://github.com/benoitgallet/ted-join-hipc22`. Accessed: March 29, 2023.

[39] David Goldberg. What every computer scientist should know about floating-point arithmetic. *ACM Computing Surveys*, 23:5–48, 1991. doi: 10.1145/103162.103163.

[40] Michael Gowanlock. Hybrid CPU/GPU Clustering in Shared Memory on the Billion Point Scale. *Proceedings of the ACM International Conference on Supercomputing*, page 35–45, 2019. doi: 10.1145/3330345.3330349.

[41] Michael Gowanlock. KNN-Joins Using a Hybrid Approach: Exploiting CPU/GPU Workload Characteristics. *Proceedings of the 12th Workshop on General Purpose Processing Using GPUs*, pages 33–42, 2019. doi: 10.1145/3300053.3319417.

[42] Michael Gowanlock. Hybrid KNN-join: Parallel nearest neighbor searches exploiting CPU and GPU architectural features. *Journal of Parallel and Distributed Computing*, 149:119–137, 2021. ISSN 0743-7315. doi: https://doi.org/10.1016/j.jpdc.2020.11.004.

[43] Michael Gowanlock and Ben Karsin. GPU Accelerated Self-join for the Distance Similarity Metric. *Proceedings of the 2018 IEEE International Parallel and Distributed Processing Symposium Workshops*, pages 477–486, 2018. doi: 10.1109/IPDPSW.2018.00095.

[44] Michael Gowanlock and Ben Karsin. GPU-Accelerated Similarity Self-Join for Multi-Dimensional Data. *Proceedings of the 15th International Workshop on Data Management on New Hardware*, 2019. doi: 10.1145/3329785.3329920. URL `https://bitbucket.org/mikegowanlock/gpu_self_join/`. Accessed: March 29, 2023.

[45] Michael Gowanlock and Ben Karsin. Accelerating the similarity self-join using the GPU. *Journal of Parallel and Distributed Computing*, 133:107 – 123, 2019. doi: 10.1016/j.jpdc.2019.06.005.

[46] Michael Gowanlock, Cody M. Rude, David Blair, Justin D. Li, and Victor Pankratius. Clustering Throughput Optimization on the GPU. *Proceedings of the 31st IEEE International Parallel and Distributed Processing Symposium*, pages 832–841, 2017. doi: 10.1109/IPDPS.2017.17.

[47] Dominik Grewe and Michael F. P. O'Boyle. A Static Task Partitioning Approach for Heterogeneous Systems Using OpenCL. *Compiler Construction*, pages 286–305, 2011. doi: 10.1007/978-3-642-19861-8_16.

[48] Antonin Guttman. R-trees: A Dynamic Index Structure for Spatial Searching. *SIGMOD Record*, 14(2):47–57, 1984. doi: 10.1145/971697.602266.

[49] Mark Harris and Kyrylo Perelygin. Cooperative Groups: Flexible CUDA Thread Programming. `https://devblogs.nvidia.com/cooperative-groups`, 2017. Accessed: March 29, 2023.

[50] Alexander Hinneburg and Daniel A. Keim. Optimal Grid-Clustering: Towards Breaking the Curse of Dimensionality in High-Dimensional Clustering. *Proceedings of the 25th International Conference on Very Large Databases*, pages 506–517, 1999.

[51] Piotr Indyk and Rajeev Motwani. Approximate Nearest Neighbors: Towards Removing the Curse of Dimensionality. *Proceedings of the 30th Annual ACM Symposium on Theory of Computing*, pages 604–613, 1998. doi: 10.1145/276698.276876.

[52] Intel. Intel Xeon Platinum 8358 Processor Specifications. `https://www.intel.com/content/www/us/en/products/sku/212282/intel-xeon-platinum-8358-processor-48m-cache-2-60-ghz/specifications.html`, 2021. Accessed: March 29, 2023.

[53] Intel. Compliance Metrics for Intel Microprocessors, Intel Xeon Processors. `https://www.intel.com/content/dam/support/us/en/documents/processors/APP-for-Intel-Xeon-Processors.pdf`, 2022. Accessed: March 29, 2023.

[54] Intel. Accelerate AI Workloads with Intel AMX. `https://www.intel.com/content/www/us/en/products/docs/accelerator-engines/advanced-matrix-extensions/ai-solution-brief.html`, 2023. Accessed: March 29, 2023.

[55] Intel. Intel Intrinsics Guide. `https://www.intel.com/content/www/us/en/docs/intrinsics-guide/index.html`, 2023. Accessed: March 29, 2023.

[56] Zhuoran Ji and Cho-Li Wang. Accelerating DBSCAN Algorithm with AI Chips for Large Datasets. *50th International Conference on Parallel Processing*, 2021. doi: 10.1145/3472456.3473518.

[57] Dmitri V. Kalashnikov. Super-EGO: fast multi-dimensional similarity join. *The VLDB Journal*, 22(4):561–585, 2013. doi: 10.1007/s00778-012-0305-7. URL `https://www.ics.uci.edu/~dvk/code/SuperEGO.html`. Accessed: March 29, 2023.

[58] J. Kim, W. Jeong, and B. Nam. Exploiting Massive Parallelism for Indexing Multi-Dimensional Datasets on the GPU. *IEEE Transactions on Parallel and Distributed Systems*, 26(8):2258–2271, 2015. doi: 10.1109/TPDS.2014.2347041.

[59] Jinwoong Kim and Beomseok Nam. Co-processing Heterogeneous Parallel Index for Multi-dimensional Datasets. *Journal of Parallel and Distributed Computing*, 113:195 – 203, 2018. doi: 10.1016/j.jpdc.2017.10.015.

[60] Jinwoong Kim, Sul-Gi Kim, and Beomseok Nam. Parallel Multi-dimensional Range Query Processing with R-trees on GPU. *Journal of Parallel and Distributed Computing*, 73(8):1195 – 1207, 2013. doi: 10.1016/j.jpdc.2013.03.015.

[61] Adam G. M. Lewis, Jackson Beall, Martin Ganahl, Markus Hauru, Shrestha Basu Mallick, and Guifre Vidal. Large-scale Distributed Linear Algebra with Tensor Processing Units. *Proceedings of the National Academy of Sciences*, 119(33), 2022. doi: 10.1073/pnas.2122762119.

[62] A. Li, S. L. Song, J. Chen, J. Li, X. Liu, N. R. Tallent, and K. J. Barker. Evaluating Modern GPU Interconnect: PCIe, NVLink, NV-SLI, NVSwitch and GPUDirect. *IEEE Transactions on Parallel and Distributed Systems*, 31(1):94–110, 2020. doi: 10.1109/TPDS.2019.2928289.

[63] Binrui Li, Shenggan Cheng, and James Lin. tcFFT: A Fast Half-Precision FFT Library for NVIDIA Tensor Cores. *2021 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 1–11, 2021. doi: 10.1109/Cluster48925.2021.00035.

[64] Swanwa Liao, Mario A Lopez, and Scott T Leutenegger. High dimensional similarity search with space filling curves. *Proceedings 17th International Conference on Data Engineering*, pages 615–622, 2001. doi: 10.1109/ICDE.2001.914876.

[65] M. D. Lieberman, J. Sankaranarayanan, and H. Samet. A Fast Similarity Join Algorithm Using Graphics Processing Units. *2008 IEEE 24th International Conference on Data Engineering*, pages 1111–1120, 2008. doi: 10.1109/ICDE.2008.4497520.

[66] Weiguo Liu, B. Schmidt, G. Voss, A. Schroder, and W. Muller-Wittig. Bio-sequence Database Scanning on a GPU. *Proceedings 20th IEEE International Parallel Distributed Processing Symposium*, 2006. ISSN 1530-2075. doi: 10.1109/IPDPS.2006.1639531.

[67] Tianjian Lu, Yi-Fan Chen, Blake Hechtman, Tao Wang, and John Anderson. Large-Scale Discrete Fourier Transform on TPUs. *IEEE Access*, 9:93422–93432, 2021. doi: 10.1109/ACCESS.2021.3092312.

[68] K. Matam, S. R. Krishna Bharadwaj Indarapu, and K. Kothapalli. Sparse Matrix-Matrix Multiplication on Modern Architectures. *19th International Conference on High Performance Computing*, pages 19–26, 2012. doi: 10.1109/HiPC.2012.6507483.

[69] MIT Haystack Observatory. Space Weather datasets. `ftp://gemini.haystack.mit.edu/pub/informatics/dbscandat.zip`. Accessed: March 29, 2023.

[70] Nvidia. Nvidia Visual Profiler User's Guide. `https://docs.nvidia.com/cuda/profiler-users-guide/index.html`. Accessed: March 29, 2023.

[71] Nvidia. Nvidia V100 Architecture Whitepaper. `https://images.nvidia.com/`

content/volta-architecture/pdf/volta-architecture-whitepaper.pdf, 2017. Accessed: March 29, 2023.

[72] Nvidia. Nvidia Turing Architecture Whitepaper. https://images.nvidia.com/aem-dam/en-zz/Solutions/design-visualization/technologies/turing-architecture/NVIDIA-Turing-Architecture-Whitepaper.pdf, 2018. Accessed: March 29, 2023.

[73] Nvidia. Nvidia A100 Architecture Whitepaper. https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/nvidia-ampere-architecture-whitepaper.pdf, 2020. Accessed: March 29, 2023.

[74] Nvidia. cuBLAS: Basic Linear Algebra on NVIDIA GPUs. https://developer.nvidia.com/cublas, 2022. Accessed: March 29, 2023.

[75] Nvidia. CUTLASS: CUDA Templates for Linear Algebra Subroutines. https://github.com/NVIDIA/cutlass, 2022. Accessed: March 29, 2023.

[76] Nvidia. Nvidia H100 Architecture Whitepaper. https://www.nvidia.com/hopper-architecture-whitepaper, 2022. Accessed: March 29, 2023.

[77] Nvidia. Double Precision GEMM Using the WMMA API. https://github.com/NVIDIA/cuda-samples, 2022. dmmaTensorCoreGemm.cu; Accessed: March 29, 2023.

[78] Nvidia. CUDA C++ Programming Guide. https://docs.nvidia.com/cuda/cuda-c-programming-guide/, 2023. Accessed: March 29, 2023.

[79] Nvidia. CUDA C++ Best Practices Guide. https://docs.nvidia.com/cuda/cuda-c-best-practices-guide/, 2023. Accessed: March 29, 2023.

[80] Nvidia. NVLink Interconnect Product Page. https://www.nvidia.com/en-us/data-center/nvlink/, 2023. Accessed: March 29, 2023.

[81] Satoshi Ohshima, Kenji Kise, Takahiro Katagiri, and Toshitsugu Yuba. Parallel Processing of Matrix Multiplication in a CPU and GPU Heterogeneous Environment. *Proceedings of the 7th International Conference on High Performance Computing for Computational Science*, pages 305–318, 2007. doi: 10.1007/978-3-540-71351-7_24.

[82] Martin Perdacher, Claudia Plant, and Christian Böhm. Cache-Oblivious High-Performance Similarity Join. *Proceedings of the 2019 International Conference on Management of Data*, page 87–104, 2019. doi: 10.1145/3299869.3319859. URL `https://gitlab.cs.univie.ac.at/martinp16cs/hilbertJoin`. Accessed: March 29, 2023.

[83] Ilya Razenshteyn, Ludwig Schmidt, Alexandr Andoni, Piotr Indyk, and Thijs Laarhoven. FALCONN: Similarity Search Over High-Dimensional Data. `https://github.com/falconn-lib/falconn`, 2015-2016. Accessed: March 29, 2023.

[84] Valerie Sarge and Michael Andersch. Tensor Core Performance on NVIDIA GPUs: The Ultimate Guide. `https://developer.nvidia.com/gtc/2020/video/s21929-vid`, 2020. Accessed: March 29, 2023.

[85] Erich Schubert, Jörg Sander, Martin Ester, Hans Peter Kriegel, and Xiaowei Xu. DBSCAN Revisited, Revisited: Why and How You Should (Still) Use DBSCAN. *ACM Transaction on Database Systems*, 42(3), 2017. doi: 10.1145/3068335.

[86] Josef Schule. Tensor Core Performance and Precision. `https://developer.nvidia.com/gtc/2019/video/s9176`, 2019. Accessed: March 29, 2023.

[87] Timos Sellis, Nick Roussopoulos, and Christos Faloutsos. The R+-Tree: A Dynamic Index For Multi-Dimensional Objects. *Proceedings of the 13th VLDB Conference*, pages 507–518, 1987.

[88] Amirhesam Shahvarani and Hans-Arno Jacobsen. A Hybrid B+-Tree As Solution for In-Memory Indexing on CPU-GPU Heterogeneous Computing Platforms. *Proceedings*

of the International Conference on Management of Data, pages 1523–1538, 2016. doi: 10.1145/2882903.2882918.

[89] Erich Strohmaier, Jack Dongarra, Horst Simon, and Martin Meuer. TOP500 Supercomputers. `https://www.top500.org/lists/top500/2022/11/`, 2022. Accessed: March 29, 2023.

[90] Stanley Tzeng, Anjul Patney, and John D Owens. Task Management for Irregular-parallel Workloads on the GPU. *Proceedings of the Conference on High Performance Graphics*, pages 29–37, 2010.

[91] Chuan Xiao, Wei Wang, Xuemin Lin, Jeffrey Xu Yu, and Guoren Wang. Efficient Similarity Joins for Near-duplicate Detection. *ACM Transactions on Database Systems*, 36(3):15:1–15:41, 2011. ISSN 0362-5915. doi: 10.1145/2000824.2000825.

[92] Zhaofeng Yan, Yuzhe Lin, Lu Peng, and Weihua Zhang. Harmonia: A High Throughput B+Tree for GPUs. *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming*, pages 133–144, 2019. doi: 10.1145/3293883.3295704.

[93] Yasuhito Ogata, Toshio Endo, Naoya Maruyama, and Satoshi Matsuoka. An Efficient, Model-based CPU-GPU Heterogeneous FFT library. *2008 IEEE International Symposium on Parallel and Distributed Processing*, pages 1–10, 2008. doi: 10.1109/IPDPS.2008.4536163.