Håvard Aspen Løvik

Top-k Spatial Join on GPU

Master's thesis in informatics
Supervisor: Kjetil Nørvåg
June 2020

Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Computer Science

NTNU
Kunnskap for en bedre verden

Håvard Aspen Løvik

Top-k Spatial Join on GPU

Master's thesis in informatics
Supervisor: Kjetil Nørvåg
June 2020

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Computer Science

NTNU
Norwegian University of
Science and Technology

Problem Description

The amount of spatial data being collected is increasing. The presence of GPS modules in common everyday tools as smart phones, cars and watches has increased the interest in making useful apps utilizing the spatial data. Both large scale commercial apps and scientific research is dependent on efficient algorithms handling this data. As the single core processor seems to be closing in on its performance limit the focus has shifted to multi-core CPUs and GPUs. In the last few decades the increase in multi-core systems availability and programmability made researchers focus on creating parallel algorithms from their traditional serial implementations. In this thesis, the aim is to investigate these new parallel spatial join methods on GPU hardware to better understand their strengths and weaknesses. What we learn will be used to design a new top-k spatial distance join algorithm for GPUs.

Supervisor: Kjetil Nørvåg

Abstract

In this thesis we will investigate spatial join queries on GPU. The specific goal is to find out if top-k spatial distance join queries are suitable for the GPU architecture.

We first discuss spatial join in general before we define the specific query we want to design an algorithm for.

Then the characteristics of traditional computer architectures are visited to understand the differences between it and the massively parallel GPU architecture. The GPU architecture is discussed to get a good understanding of the execution model. Important aspects that utilize the hardware to its full potential is also discussed.

Next, two common spatial indexes called R-tree and uniform grid are discussed. This is to give context to our choice of the uniform grid as the spatial index for the top-k spatial distance join query.

To get a better understanding of how previous research have handled spatial join on GPU we choose four different spatial join methods to investigate. The purpose is to learn which techniques works well for GPU and what we need to be cautious of when designing our own algorithm.

Lastly the top-k spatial distance join algorithm is design and benchmarked against a simple parallel approach. The results show that the spatial distance join part of the problem is well suited for GPU parallelization. Adding the top-k part introduce significant challenges that leaves none of our benchmarked methods without significant drawbacks to the hardware utilization.

Sammendrag

Oppgaven utforsker romlige sammensettingsspørringer utført på GPU. Målet er å utforske om en top-k romlig sammensettingsspørring gir mening å utføre på GPU.

Først introduseres romlige sammensettninger generelt før den spesifikke spørringen blir definert. Så diskuteres karakteristikken til tradisjonelle datamaskinarkitekturer for å gi kontekst til hvorfor GPU arkitekturen er som den er og hvordan man effektivt kan programmere den.

Neste del av oppgaven diskuterer to forskjellige romlige søkestrukturer som heter R-trær og uniforme rutenett. Dette gir kontekst til valget for en romlig søkestruktur.

For å få en bedre forståelse av hvordan tidligere forskning har håndtert romlige sammensettninger på GPU, undersøker vi fire forskjellige metoder. Målet er å finne ut hvilke teknikker som passer bra til GPU arkitekturen og hvilke aspekter vi må være oppmerksom på når vi desinger vår egen algoritme.

Til slutt blir den nye algoritmen designet og sammenlignet med en enkel parallel algoritme på GPU. Resultatene viser at romlige sammensettningsspørringer alene passer bra til GPU parallelisering. Om man legger til top-k delen av problemet, støter vi på signifikante utfordringer hvor ingen av metodene vi testet ungikk store utfordringer med å utnytte GPU maskinvaren til sitt fulle potensial.

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

1.1 Motivation and related work

Spatial data collection is increasing. We have Global Positioning System (GPS) modules in smart phones, computers, tablets, watches, cars etc.. Anywhere we go there is a potential of generating spatial data. This has lead to an increased interest in applications using this data both in commercial and scientific fields.

The join operator is essential when dealing with spatial data. The spatial join operator can for example answer queries asking which objects intersects, are in close proximity to each other or are similar. When we combine the join operator with the top-k query we can filter out irrelevant results based on our own ranking/scoring function. Some of the use cases that are mentioned in [1], [2] and [3] for spatial join queries are listed below:

- When traveling to a new city the hotel and restaurant quality may be important. The top-k spatial join query can give you the top five combinations of hotels and restaurant within a kilometer distance from each other based on the user ratings from other customers.

- Top-k spatial distance join can also be used in bioinformatics. Identifying pairs of amino acids that exhibit "good" properties that contribute to the stability of a protein is one use case. The properties of amino acids can be assigned scores and the top-k spatial join query can identify pairs of amino acids that are close to each other with respect to their 3D location. The pairs of amino acids with the highest scores within a certain distance can contribute best to the stability of a protein.

- Often different spatial datasets need to be joined to derive new information and knowledge to support decision making. For example, GPS traces can be better interpreted when aligned with urban infrastructures, such as road networks and Point of Interests (POIs), through spatial joins. As spatial datasets are getting increasingly larger, techniques for high-performance spatial join processing on commodity and inexpensive parallel hardware become crucial in addressing the "BigData" challenge.

Previous research has focused on making algorithms for the Central Processing Unit (CPU). This has been going for several decades and some techniques are well established. For example the R-tree is a well established indexing structure for spatial data that is used in many of the traditional spatial join algorithms.

Spatial join on Graphical Processing Unit (GPU) is by contrast a relatively new field. One problem researchers have faced is that traditional techniques that rely on hierarchical structures such as the R-tree [4] are not inherently suited for GPU. There is done research on trying to implement efficient R-tree structures on GPU [5], [6], [7]. Their focus is on trying to minimize the effect of the branching nature of the tree structure and fast construction. They show significant improvements compared to the CPU baseline methods they compared to. Still there seems to be hesitation to use R-tree structures for spatial join on GPU.

Much of the previous research on spatial join on GPU have tried to use other simpler structures [8], [9], [10] and [11]. In [8] they use a uniform grid structure to implement a spatial intersection join between boxes. The method is made for real time intersection join on moving objects. The performance shows significant improvements over the previous state of the art method.

Grid structures are also used in [9] they use a grid structure for distance self-join. Since the query is based on a distance measure the grid cells are sized according to this distance allowing for an efficient distance comparisons of points.

In [10] they make a system for spatial join between polygons on GPU. They do not use an index structure at all. The common reasoning between these articles are that heavy indexes such as R-trees have problems fitting well in a GPU context.

The article most relevant to the specific query we are investigating is [1]. They implement a top-k spatial distance join operation that is based both on the score and spatial attributes for CPU. They use an R-tree as the spatial index and a min-heap to hold candidate results. They also rely on a common threshold based on the previously found scores to stop early. These are all techniques that do not fit well on a massively parallel architecture. We will therefore have to think differently when solving this problem for GPUs. To their knowledge they were the first to implement such an algorithm. In [12] they solve a similar problem but their solution was restricted to attributes based on probabilities and the aggregation function had to be product.

To our knowledge they still are the only ones researching this problem. The most similar research we could find on GPU was [13] which investigated top-k trajectory similarity join on GPU. That means that there is no attempts to solve this problem on GPU. Therefor the top-k spatial distance join problem is an interesting problem to create a new algorithm for on GPU.

The only research we could find that solved top-k on GPU was [14]. They solve a pure top-k problem using a method based on radix sort where they find ways to stop early since sorting all data may be unnecessary.

In addition the authors of paper [15] that go through interesting directions for future research on spatial joins specifically mentions that the ε distance join is an interesting problem to try to parallelize.

This motivates the goal of this thesis that is to investigate which techniques that are suited for top-k spatial distance join on GPU and use this knowledge to design a new top-k spatial join algorithm designed for the highly parallel GPU architecture.

1.2 Research questions

The main question this thesis tries to answer is how the top-k spatial distance join query can be efficiently executed on the GPU. This implies that we can end up with and answer that tells us that GPUs are not suited. To be able to answer this there are some important questions that needs to be answered.

One sub-research question that have to be answered is how we can structure the data for efficient utilization. Spatial indexes are commonly used by spatial algorithms to speed up search and manipulation of the data. Finding which spatial indexes that benefits from GPU parallelization will be an important step on the way to an efficient algorithm for spatial joins on GPU.

The second sub research question will ask how the top-k part of the query can be efficiently executed on the GPU. Since the high speed GPU memory is limited and little research is done on top-k queries on GPU this is an important question to answer. The research questions are formally defined as:
RQ1: Are GPUs suited for top-k spatial distance join queries?

- **SRQ1**: Which spatial indexes and data types if any are suited for spatial join on GPU?
- **SRQ2**: How can top-k be efficiently implemented in the GPU memory architecture?

Hopefully by answering these questions it can give us knowledge also outside of the specific query we have chosen.

Chapter 2

Background

2.1 Database queries

Database queries are requests sent to a Database Management System (DBMS) written in a high level query language such as the Structured Query Language (SQL). The queries are processed by the DBMS and the return value is a result set or an error message. A very common query that will be the focus of this thesis is the join query denoted by R $\bowtie_{A=B}$ S, where R and S represents two tables and A and B are respective attributes.

The overall query processing flow is described in Figure 2.1.

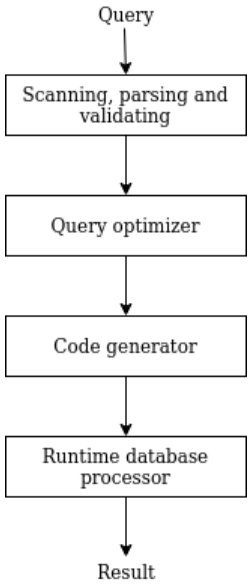
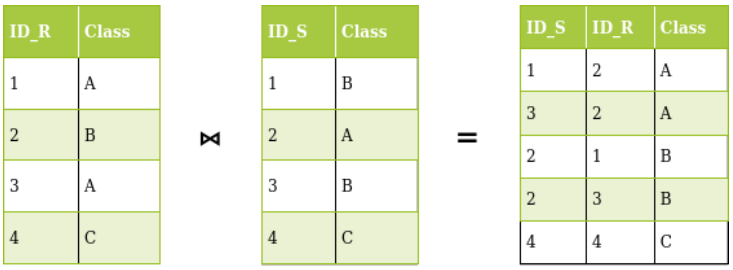


Figure 2.1: The steps involved in a typical database query process.

The first step is to translate the query into a format usable for the DBMS. This involves scanning the query for key words specific to the querying language, parsing the query to see if it has valid syntax and validating to see if attributes and relations makes sense. The result is an internal representation of the query often represented by a Directed Acyclic Graph (DAG) that is guaranteed to be a valid query.

The DBMS can then start planing how the query will be executed. This is often called the query optimization step. The goal of this step is to find a fitting execution strategy for the given



ID_R	Class
1	A
2	B
3	A
4	C

⋈

ID_S	Class
1	B
2	A
3	B
4	C

=

ID_S	ID_R	Class
1	2	A
3	2	A
2	1	B
2	3	B
4	4	C

Figure 2.2: A simple join between two tables where the equality predicate is on the class attribute.

query. Query optimization may be a bit miss leading since the DBMS does not necessarily have all sources of information that is needed to make the optimal plan. Finding an optimal plan would in many cases be to time consuming as well. There may also be information needed that is not available, for example the distribution of the data. The DBMSs job is therefore to quickly find a good execution plan with the given information.

Based on this plan the DBMS generates the code required to execute the query in the runtime database processor. After the query is executed the result is returned to the user. In this thesis the main focus will be on making an efficient execution step for the top-k spatial distance join query.

2.1.1 Join and spatial join queries

There are many different types of join queries. The basic idea is that two or more tables in a relational database can be joined together based on a join predicate. Join means that tuples from two or more tables are combined into a single longer tuple. The most common join method involves a join predicate checking for equality between two attributes. The equality join operation between two tables are illustrated in Figure 2.2.

In this thesis we will be investigating the intersection and distance join queries. The intersection join query will be using boxes as the spatial objects and the distance join query will be using points. The reasoning behind this choice is that they have a wide array of use cases and are the most common queries to find in previous research [15]. Queries on polygons are also common but much of the research focus on finding efficient methods for polygons intersection. This is outside of the scope of this thesis, but the filtering part of those algorithms is still relevant and will therefore be considered as well.

The common concern when processing spatial queries is to find some way to decrease the number of comparisons needed when executing the query. As demonstrated in Algorithm 1 this is an $\mathcal{O}(n^2)$ algorithm. This is especially important for algorithms that needs complex checks to verify the join predicate. For example intersection join on polygons have complex checking algorithms so the best solution is to avoid as many comparisons as possible. An expensive filtering step to reduce the candidate pairs can most often be justified.

Figure 2.3 describes the typical spatial join process. R-trees(Section 3.2) and grid structures as uniform grids(Section 3.3) are commonly used structures for spatial joins. The benefit of using indexes such as R-trees on spatial joins is that finding and comparing spatial objects can be done very efficient as long as you have already made the spatial index. The disadvantage is that the creation of such indexes can be expensive, especially if they are never reused. Methods that do not use indexes are also developed. The sweeping line method is an example of this [16]. The indexes are used to make the filtering step be faster.

The main purpose of filtering is to generate potential candidate pairs for the refinement step. This is because the refinement is so slow that even with two extra steps of indexing and filtering we still save time. The filtering step most commonly use MBBs as defined in Section 3 to simplify the geometric shapes. This is why the filtering step of the polygon intersection queries is relevant to box intersection queries.

The refinement step does the last thorough check to produce the final pairs. This step depends on the predicate and data used in the query. For example when intersection on polygons is the predicate edge intersection and point in polygon test can be done. This is a very expensive step.

It is important to note that the need of a filter step depends on the cost of the actual join predicate check. For example for point data where we want to find pairs of points within a given distance of each other the filtering and refinement steps are combined into one. This is because the distance check is so fast that it often does not make sense to do it in two steps. A common way to execute this kind of algorithm is to index the points in a way that reduces the number of comparisons drastically, and then just execute the actual comparisons using this index.

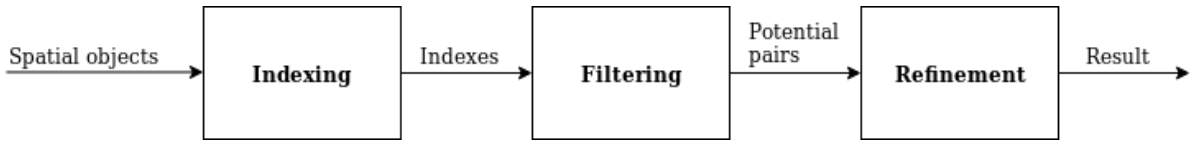


Figure 2.3: The typical process of a spatial join algorithm.

2.1.2 Top-k spatial distance join

In this section the top-k Spatial Distance Join (top-k SDJ) problem is introduced. The problem will be formally defined and an example of a naive serial algorithm will be presented. This is the problem we will try to design an efficient algorithm for on GPU in Chapter 5.

This is a very similar definition of the problem as in [1]. We start out by having two sets R and S. These two sets contain spatial coordinates within $\mathbb{R}^n$ and a score. We do not require the sets to be sorted by score as in [1]. The goal of the top-k SDJ algorithm is to find the top-k combinations of two elements, one from each set, which have the highest combined score based on a scoring function γ. Our scoring function does not have to be monotone as in [1]. The objects also need to satisfy the join predicate which is that the two objects needs to be within ε distance from each other. The result is a set C with length K that contains the top scoring element combinations. The formal definition can be found in Definition 1.

Definition 1. (Top-k Spatial Distance Join). Let $R = \{r_1, r_2 ... r_{m-1}, r_m\}$ and $S = \{s_1, s_2 ... s_{n-1}, s_n\}$ be two sets of spatial objects, k be a positive integer $k \in Z^+$, ε be a distance threshold $\varepsilon \in \mathbb{R}_{>0}$, $\gamma(r,s)$ be a monotone function $\gamma : \mathbb{R} \times \mathbb{R} \to \mathbb{R}$ and $dist$ be a distance measure. The top-k SDJ query will return a set C of length k with spatial object pairs $x = (r,s)$ were $dist(r,s) \leq \varepsilon$. No other pair x' in $R \times S - C$ have a higher score than any object in C, $\forall x \in C(\neg \exists x' \in R \times S - C | \gamma(x) < \gamma(x'))$

The distance measure in this algorithm can be any general distance called a Minkowski distance L_t. The distance is measured between two points in $\mathbb{R}^n$. For t = 2(L_2) it is called euclidean

distance, for t = 1(L_1) it is called the Manhattan distance and for t = ∞(L_∞) it is called the Maximum distance. Which distance metric to use depends on the application. In this thesis we will use the L_2 euclidean distance.

A naive approach to solving this problem would be slow. One approach is to compare all objects with each other to get the pairs that are closer than ε. We could then calculate the score γ of all the pairs that are within ε distance and sort them in descending order. Taking the top-k elements from the sorted list would give the correct result. The worst-case running time would be $\mathcal{O}(n^2)$ for the distance comparison, and $\mathcal{O}(n\log(n))$ to sort based on γ assuming an optimal comparison sort algorithm and $\mathcal{O}(1)$ to fetch the top-k. The running time would then be dominated by the distance comparisons so $\mathcal{O}(n^2)$ would be the upper bound. Algorithm 1.

Algorithm 1: Naive Top-k SDJ

```
   input  : R, S, γ, ε, k
   output: A set C with the top-k object combinations
 1 Pairs ← ∅ ;
 2 foreach r ∈ R do
 3 |   foreach s ∈ S do
 4 |   |   if dist(r, s) ≤ ε then
 5 |   |   |   Pairs := Pairs ∩ (r, s);
 6 |   end
 7 end
 8 Descending ← Sort(Pairs, γ);
 9 C ← GetTop-k(Descending, k);
10 return C ;
```

2.2 Traditional computer architecture and parallelization

This section is meant to give the background knowledge needed to understand why the GPU architecture is designed the way it is. Section 2.2.1 describes the von Neumann architecture before Section 2.2.3 describes the overall process of solving a problem in parallel. The GPU architecture is then discussed in Section 2.3 before we discuss how to program them efficiently in Section 2.3.2.

2.2.1 von Neumann architecture

The von Neumann architecture [17] is a classical computer architecture by mathematician and physicist John von Neumann. It is a simple concept consisting of a central processing unit and a memory module. The CPU have two sub-modules called the Control Unit (CU) and the Arithmetic and Logical Unit (ALU). The CPU and memory module is interconnected allowing the CPU to read and write from memory Figure 2.4.

The control unit is responsible for knowing which instruction should be executed. It does this by keeping track of a Program Counter (PC) which holds the address of the next instruction. The ALU is responsible for executing these instructions.

Inside the CPU there are small memory modules called registers. These registers are relatively fast and mainly used for keeping track of the program logic or saving intermediate values

during computations by the ALU. They are only used for storing data and instructions necessary to execute the current program. The program counter is an example of a register.

The main memory module outside the CPU is much larger than the registers in the CPU. The memory module stores data and instructions needed to execute the program. For the CPU to be able to access this memory it uses the interconnections between the two modules often called the bus.

The memory module is much slower than the CPU registers. Since the CPU needs to access this memory for data and instructions it is slowed down. This separation of memory and CPU is called the von Neumann bottleneck, since the interconnection determines how fast instructions and data can be accessed hence affecting the execution time.

To improve upon this architecture there have been a lot of research on how to better utilize the CPU for efficient computations and finding smarter ways to access memory to overcome the von Neumann bottleneck.

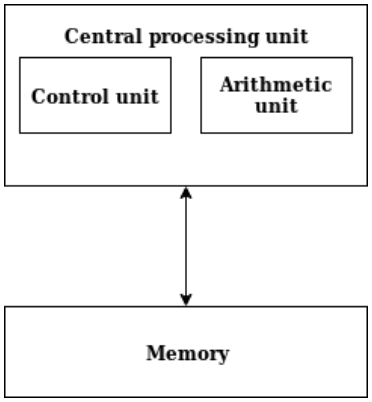


Figure 2.4: The von Neumann architecture.

2.2.2 Memory hierarchy

The memory hierarchy is a concept developed to overcome the von Neumann bottleneck. The idea is that we can limit the effect of the slow memory access by placing smaller but much faster memory modules between the memory and the CPU. These memory modules are called cache modules. They are ordered in a hierarchy where the smallest and fastest module is close to the CPU. When moving further away form the CPU the cache modules gets larger and slower. Figure 2.5 shows an example of a typical memory hierarchy on a modern computer. There are three layers where the first layer can hold some kilobytes of data while the third layer can hold a few megabytes. The main memory on modern computers are now often tens of gigabytes. This means that there is a very small percentage of the data in the main memory module that can be stored in cache modules.

To chose what data is stored in the cache modules we need to look at the memory usage of most programs. When executing a program most instructions happen in sequence. Programs and data are stored consecutively in memory. This means that when executing a program we can be pretty certain that after one instruction the next instruction will be the one laying consecutively in memory. This is called the spatial locality cache principle.

Branching may contradict this but its found that branching instructions makes up a very small portion of typical programs. The spatial locality principle also holds for data. Then we

access a memory location we want to cache the surrounding data since they will most likely be accessed next. It may vary how much data and for how long the data will be stored in cache.

A good example of utilizing the spatial cache principle is when we access a 2D matrix in memory. If we store the rows consecutively in memory we want to loop over all items/columns in that row before moving to the next. This allows the cache to cache the rows consecutively in memory. If we had accessed it in a column mayor fashion we would no be able to do the same as each access would have a whole row of data between them.

The temporal locality principle is similar. Instead of focusing on the spacial information it focus on the time aspect. When accessing data it will likely be accessed again in the near future. If we cache the accessed data we can reduce the amounts of times we need to access the slow main memory data.

To take advantage of these cache principles we fetch data and instructions in whats called cache blocks or cache lines. The idea is that since the data laying next to the data we want to access is so likely to be accessed next we can fetch more than only one data item or instruction per operation. Typically 8 or 16 times more data can be accessed in one transaction and cached. How much will depend on the architecture. If the next memory fetch instruction finds the data or instruction it is looking for in cache it is called a cache hit. If it does not it is called a cache miss.

The caching principle is also very important in modern GPUs where the execution cores can execute operations much faster than the main memory module can keep up with. Memory access latency is a big challenge when developing efficient GPU algorithms. Good memory access patterns to maximize cache usage is an important method for better performance. More on this in Chapter 2.3.

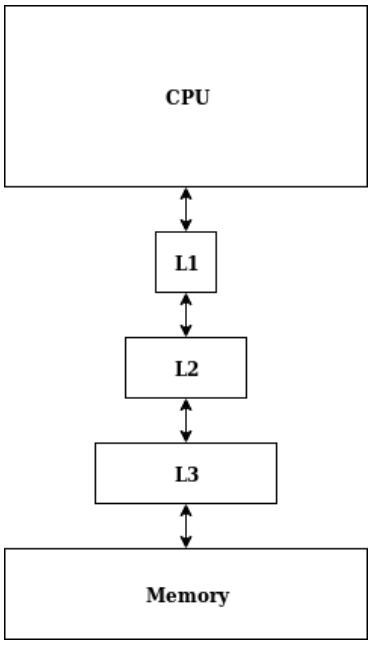


Figure 2.5: The typical memory hierarchy in modern computers.

2.2.3 Parallel computing

For a long time software was written in serial. This worked well when the performance increase was mainly driven by Mores law and more efficient algorithms. When the transistors got smaller and smaller the heat density also increased. To overcome this problem the focus on making faster cores shifted to making multi-core processors instead. This lead to new challenges where the old serial algorithms had to be revise to work on parallel systems to utilize the core. To do this we can rewrite the serial algorithm to a complete new parallel algorithm or we can write a translation program that automates the process. Research on automatic translators have shown to have limited effects. The problem is that writing a parallel program is not trivial and often needs complete redesign of the algorithm to give good results. The translators currently does not have the capability to do this and can only recognize common serial patterns that can be translated. This is not sufficient for a good parallel implementation as the original serial sequence of tasks may be terribly inefficient on the parallel processor. A complete rewrite currently needs a human developer.

Writing a parallel program is done by carefully considering the parallel architecture and dividing the problem into tasks that can be done in parallel. Problems that have independent sub problems where each sub problem can be solved individually are perfect for parallel algorithms. This is why for example graphics processing benefit so much from parallel execution since each individual pixel can be processed individually.

There are two main schemes in task division called task parallelism and data parallelism. Task parallelism is dividing different task out for all the processor cores where the cores may process the same or different data.

A simple example of task parallelism: Consider a car factory. Along the assembly line there are 5 robots that can work simultaneously. Each robot is responsible for assembling $\frac{1}{5}$ of the car. The robots will represent the cores in a multi-core system. Each core does different tasks in parallel on the same data, hence task parallel.

A simple example of data parallelism: Consider the same car factory as the last example, but instead of one assembly line we have 5 lines doing the same work but with one robot per line. The cars would be split onto the 5 assembly lines. Here the individual assembly lines represent a single core. The same tasks are performed in parallel but on different cars or 'data', hence data parallel.

What technique that is best is problem and hardware dependent. For this problem the most efficient method would depend on how the work was divided among the workers or robots in the task parallel method. If we assume the tasks parallel process was divided so there was exactly as much work for each worker it would be as fast as the data parallel method. If not we may get the problem that one worker finish before the others and need to wait doing nothing. Redistributing this worker to help the other workers may help but will require communication overhead. One disadvantage to data parallelism is that each core may get very complex as each core needs to execute the whole process by itself. If there is limited register space per core this may become a problem.

GPUs are perfect for data parallel processing as explained in Section 2.3. Using Flynn's taxonomy GPUs best fit the Single Instruction Multiple Data (SIMD) execution model but since each core have multiple threads it is often called Single Instruction Multiple Threads (SIMT). It is therefore data parallel algorithms that will be the focus of this thesis.

2.3 GPU architecture and CUDA

The Graphics Processing Unit (GPU) was originally created to accelerate graphical processes. Computer games have traditionally been computationally heavy where complex graphical scenes must be rendered up to hundreds of times per second. The basic building blocks of computer graphics are simple primitives (usually triangles) represented by floating point numbers. To move the viewpoint around the scene we apply combinations of simple affine transformations to these primitives. This means that millions of floating point computations need to be done every time we want to render a scene. This has lead to games being an important pushing force in the development of GPUs since they often require high frame rates and detailed scenes. In the last decades as the GPUs performance have steadily increased and they have turned into general purpose GPUs. This has lead other areas to also show interest in the highly parallel architecture. Neural networks have recently become very popular as the generation of neural network models benefit greatly from the parallel nature of the GPU. All discussions on GPUs will assume an NVIDIA GPU. The next Section 2.3.1 will describe the GPU architecture while Section 2.3.2 will explain how it is programmed using Compute Unified Device Architecture (CUDA).

2.3.1 Architecture

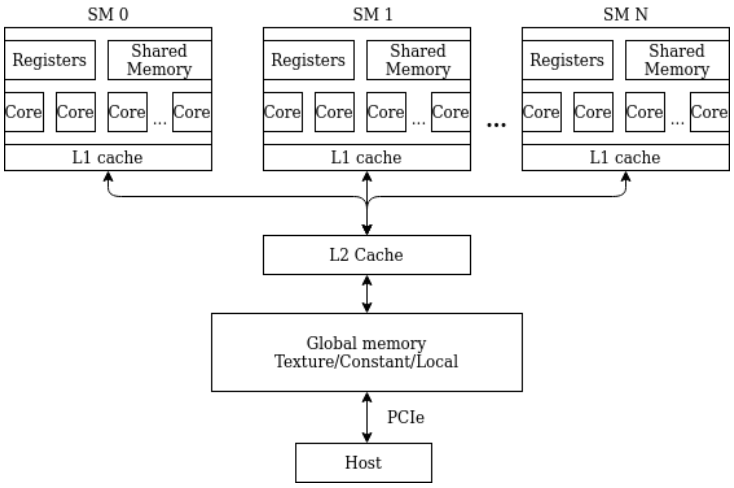


Figure 2.6: A simple diagram of the typical NVIDIA GPU architecture seen from a programmers perspective.

Figure 2.6 gives a simplified overview of a typical GPU architecture from the programmers perspective. The GPU consists of a collection of Streaming Multiprocessors (SM). Each SM has multiple cores for executing threads. The cores inside the SMs are called CUDA cores. These cores are essentially a Floating Point Unit (FPU) designed for efficient floating point operations and an ALU. There are also memory modules on each SM that are described in the memory list below. Each SM have a large amount of cores. The Tesla P100 (see Table 6.4) we will be using in this thesis have 56 SMs with 64 cores combining to a total of 3584 cores. This is a massive parallelization increase compared to multi-core CPUs. The GPU does have slower cores than CPUs often ranging between 1 to 2 GHZ but the large amount of cores makes the GPU perfect for tasks that have independent sub-problems.

To schedule work each SM has its own scheduler called a warp scheduler. Warps are a collection of 32 threads executed synchronously. Further details on warps can be found in the next Section 2.3.2. The warp scheduler can actively switch between the warps that are executed between each clock cycle. This is essential for techniques as latency hiding. Other than the already mentioned parts the most important aspects of the GPU is the memory structure. The memory structure on a GPU is as follows:

- Global memory: The large Dynamic Random Access Memory (DRAM) memory banks outside the SM's that holds most of the memory space available to the GPU. Global memory is often used when memory space is allocated on the GPU device from the host before it is copied over from the computers main memory. Global memory is slow compared to the other memory banks on the GPU, so you want to minimize global memory usage.

- Constant memory: Is also resident on the DRAM memory banks. Constant memory is optimized for constant values and can be accessed more efficiently than if 'normal' global memory was used. Constant memory is also cached. __constant__ is used to declare constant memory.

- Texture memory: Texture memory is also allocated on the DRAM memory banks and share the same space as global memory. The difference is that texture memory access is highly optimized for 2D access patterns. This was originally done to accelerate texture access for the graphics pipeline but have proven useful for general purpose computing as well.

- L1 and L2 cache: since global memory is relatively slow there are multiple layers of cache between it and the SMs. L2 cache is a shared layer of memory between the SMs and the global memory. It is much smaller than global memory but significantly faster. L1 cache is a local per SM memory chip that is even smaller and faster than L2 cache. L1 cache is also often called "Unified Memory". The sizes of these caches can be very small compared to the size of the global memory. The difference can be in the order of a few MB to more than 10 GB.

  It is important to note that not all architectures have both L1 and L2 cache. This will depend on which architecture the GPU use. The caches are typically not available to the developer and is handled by the GPU itself. The programmer can although use smart access pattern to utilize the caches as much as possible.

- Shared memory: Each SM has its own shared memory chip that is local on the SM. This chip is significantly faster than the global memory but it is quite small. Shared memory is shared by all the blocks currently resident on the SM. If you have to access global memory multiple times you may want to use shared memory as a buffer to minimize the times you have to access global memory. Shared memory can be used by specifying __share__ when declaring.

  The shared memory is organized into 32 banks, one for each thread in a warp. This is so threads can access shared memory in parallel. Storing data in shared memory that spreads the out on these banks is therefor important. If not the share memory access is serialized. Shared memory can also be used for communication purposes between threads in a block.

- Registers: Registers are the memory space with the lowest access time. They can be used by the threads to for example store the increment variable in a for loop. If the registers

get full the treads may store data in local memory. It is the CUDA compiler that decides if it wants to use registers or local memory.

- Local memory: It is a part of the global memory assigned to large thread local data. That is if a thread wants to allocate space for data that is too large for the registers to handle it is allocated in local memory instead. Local memory is much slower than registers but lets programs threads use allocate more memory than available in the registers.

2.3.2 CUDA

Compute Unified Device Architecture (CUDA) is a parallel computing platform and Application Programming Interface (API) created by NVIDIA. The following section will not discuss specific API calls but rather explanation what we need to think of when writing a program for GPU.

A common CUDA program follows these steps:

1. Allocate memory space on the GPU.
2. Copy data from the host machine to the GPUs global, constant or texture memory.
3. The host machine specifies how GPU threads should be structured when executing a kernel. A kernel is a program running on the GPU.
4. The device finish executing the kernel and copies data back to the host machine.

To understand how to program using CUDA it is important to understand how threads are executed. Warps are a collection of 32 threads that are executed simultaneously on the CUDA cores in an SM. Each warp is taken from three dimensional blocks of threads simply called blocks. All blocks are organized into a three dimensional grid. This structure is used so individual threads can be indexed using their x, y and z coordinates. When a GPU kernel is executed the grid is turned into a queue of blocks. Each block is waiting to be assigned to a SM with free capacity. All the warps in the block need to be finished before a block can be freed from the SM. When all warps in all the blocks are finished executing the kernel is finished.

So what determines how many blocks can be active on a SM? There is a max limit to warps, threads and blocks. On a NVIDIA Tesla P100 that will be used for benchmarking in this thesis the maximum is 32 blocks, 64 warps and 2048 threads. These are only theoretical maximums. If we want to know the actual number that can be assigned it is a bit more complicated.

When a block is assigned to a SM all its registers and shared memory that the block will use is allocated. This means that if you have 65536 registers available on the SM and each block have 16 warps where each thread uses 32 registers we get that $\frac{65536}{(16*32)*32} = 4$ blocks can be assigned to the SM. Note that if 4 blocks were assigned to the SM that would result in 2048 threads which is the maximum. Decreasing the register usage per thread would not increase the number of blocks that could be assigned since we have already hit the max thread limit. The same principle is true for shared memory as well.

Since register and shared memory space is allocated per whole block we get an interesting scenario if we use one more register per thread. Then we get $\frac{65536}{(16*32))*(32+1)} = 3.88$. This causes only three blocks to be assigned to the SM resulting in 14848 registers to be unused. An illustration of this can be seen in Figure 2.7. One technique that makes the impact of many registers per thread smaller is to reduce the number of warps in a block. By doing this we let

the blocks be allocated on a fine grained level that will reduce the number of unused registers significantly. It also makes more warps available that is good for latency hiding.

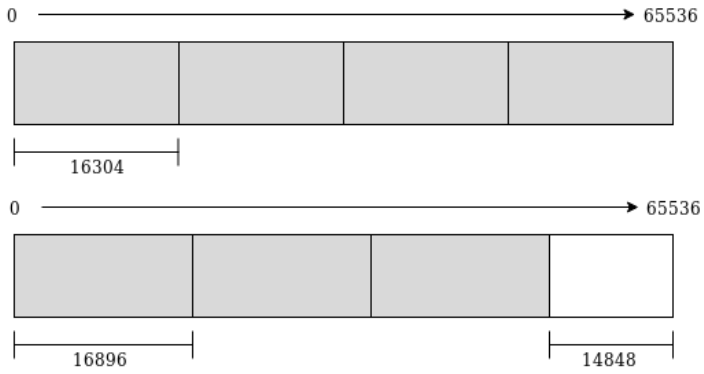


Figure 2.7: (Top) Register usage per thread is 32 with 16 warps per block. This fully utilize the register space. (Bottom) Register usage is 33 with the same block dimensions. Now we can only fit 3 blocks which leaves 14848 registers unused.

Register spilling is another effect of using too many registers per thread. If the compiler finds that the data allocated by threads is too large to fit in registers it may need decide to save it in local memory instead. As explained before the local memory is much slower than registers an may therefore degrade the performance of the kernel significantly.

Memory space on the SM is allocated per block because it lets the warp scheduler switch between active threads instantaneously. The warp schedulers task is to schedule the warp executions as efficiently as possible. One technique the warp scheduler use is to switch between warps when they stall. A very common causes for stalling is memory access where the warp needs to wait for data from global memory. The scheduler switch to another warp that is not stalling while the warp is waiting. This will essential hide the memory access delays if there are enough warps available. What allows us to use this technique is that the registers and shared memory are not deallocated when the warp is deactivated. Since the memory space for all warps (threads) got allocated at the start when the block was assigned to the SM the only thing the scheduler needs to do is to tell which warps should be active. It is then also obvious that having a high amount of warps available to switch between is beneficial for the latency hiding technique. Figure 2.8 shows an example of latency hiding.

We do therefore also need to be careful of how much register and shared memory space we use. Using too much will lead to few blocks being resident on the SM which in turn leads to low occupancy.

To reduce the impact of memory requests the GPU tries to make as few transactions for data as possible to minimize DRAM bandwidth. One transaction to the global memory fetches 32, 64 or 128 bytes of data per transaction. The transactions are using these sizes because since a warp can then use only one transaction to fetch one data item per thread. For example a warp can use one transaction to fetch 32 floats of 4 bytes each. The GPU trying to minimize the memory transfers into as few transactions as possible is called memory coalescing. Figure 2.9 shows shows a optimal access pattern where only one transaction is needed per warp.

When data is stored in global memory it is aligned to fit into these transaction sizes. This means that if you were to access data that is not aligned with the transactions sizes we would need multiple transactions to access the data requested by a warp. This would be bad if the transactions were not cached. This was the case on old GPUs with compute capability less than

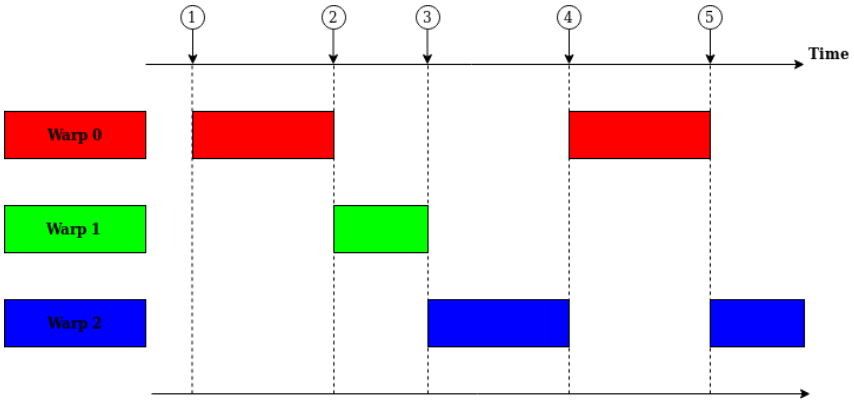


Figure 2.8: (1) Scheduler starts executing warp 0. (2) Warp 0 executes code with memory latency, scheduler switches to executing warp 1. (3) Warp 1 executes code with memory latency. Switches to warp 2. (4) While warp 1 and 2 have executed the data warp 0 was waiting for is now ready. So when warp 3 meets a memory latency the scheduler switches to warp 0.(5) The data warp 2 was waiting for is now ready. When warp 0 hits a new memory latency the scheduler switches to warp 2. Warp 1 is still waiting for the memory request to finish.

2. On modern GPUs as the Tesla P100 the memory transactions are cached in L1 cache. If we execute multiple transactions the cache lines will be available in cache for the next warp to access making the effect of miss aligned access pattern having performance closed to an aligned pattern. The middle diagram in Figure 2.9 shows miss aligned access. An access pattern that will always be problematic is a strided access pattern shown in the bottom diagram in Figure 2.9. With strided access each tread in the warp wants to access a separate memory location contained in separate transactions. Since the cache cannot does not help with this pattern the resulting performance is bad on all architectures.

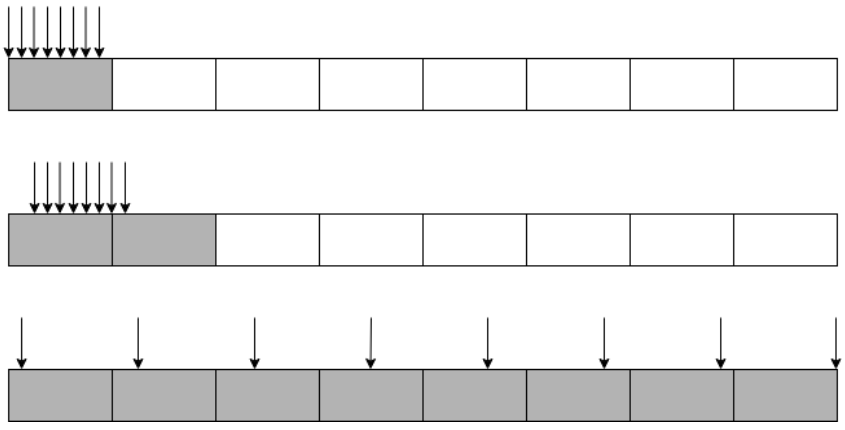


Figure 2.9: (Top) An ideal access pattern where all memory requests resides within the same transaction unit. (Middle) Memory access pattern where all accesses are sequential but offset. The performance of an offset access pattern will average out to be very close to an optimal access pattern on modern GPUs. (Bottom) A strided access pattern where each request would require a separate transaction. Will perform bad on all GPU architectures.

Another important thing to note in addition to block sizes and register memory usage is thread divergence. Thread divergence is a problem where we have branching internally in a warp. This can happen if some threads in a warp qualify an if-statement while some do not.

Since all threads have to do the same instruction the threads that did not qualify the if-statement have to be idle while the other threads run(and opposite for the other threads). This can lead to many threads being idle for large portions of the program. Therefor we want our code to have as few branches as possible.

2.3.3 GPU evolution

The GPU architectures evolve over time as better performance is needed. For example GPUs made using the Fermi architecture in 2010 have different approaches to problems than the GPUs made with the Pascal architecture in 2016. NVIDIA use the term compute capability do distinguish between the different architectures. For example the Fermi architecture have compute capability 2 while the Pascal architecture have compute capability 6. There may also be variations within an architecture.

While the same basic concepts apply through all the architectures, some small changes may lead to differences in the algorithm design. For example the Fermi architecture did not have support for dynamic parallelization, that was introduced with the Kepler architecture succeeding Fermi. The dynamic parallelization made it much easier to implement hierarchical structures and recursive algorithms on the GPU. Previous architectures only allowed the CPU to execute kernels on the GPU. When dynamic parallelization was implemented the GPU could execute more kernels itself. Another aspect that also use to change between architectures are how much memory is available in the different modules and the cache layout.

These architectural changes lead to changes in how the developers and researchers have to think. It is therefore important that we are aware of which hardware the algorithm will be running on and which limitations or improvements that can occur when running on other architectures. In this thesis we are using the Tesla P100 GPU based on the Pascal architecture.

Chapter 3

Spatial indexes

This chapter will cover two common spatial data structures that are used to index spatial data. Each structure will be shortly described and their performance will be discussed in a GPU context. The main purpose of this section is to show which structures are suited for spatial join algorithms on GPU, and specifically a structure that can be used for the top-k SDJ algorithm. Section 3.2 covers R-trees while Section 3.3 covers the uniform grid structure.

A spatial index allows algorithms dealing with spatial objects to access them efficiently compared to non structured data which would have to be accessed by a sequential search. Finding a good spatial structure is both problem dependent and execution model dependent. Examples of commonly used indexes for solving spatial problems are R-trees and uniform grids. All spatial indexes discussed will consider objects in a real coordinate space of n-dimensions, $\mathbb{R}^n$.

Definition 2. (Spatial object) Spatial objects can form any shape in $\mathbb{R}^n$. The object X is a set of points subject to $X \subseteq \mathbb{R}^n$.

Most spatial indexes use Minimum Bounding Boxes (MBBs) to represent spatial objects. Figure 3.1 shows examples of MBBs for some common shapes. The MBB is an axis aligned box with the smallest measure which still covers all points in the object. The measures can for example be area in two dimensions or volume in three dimensions.

Definition 3. (MBB) A minimum bounding box B enclosing an object X such that $X \subseteq M$ and no other box B' satisfying this has a smaller measure $|B'| < |B|$.

MBBs are efficient in space usage since they only use two points to represent shapes. The first point is the minimum of all n dimensions and the second is the maximum. By using only two points to represent an object we can significantly speed up for example intersection between two polygon objects.

We start by initially only checking if their MBBs intersects. The MBBs make this an efficient process in both space and time. If we have an intersection between the MBBs a more thorough check can be used to find out if the objects actually intersects. This method is a crucial part of why spatial indexes are so efficient.

3.1 Curse of dimensionality

The curse of dimensionality is a common problem in fields that process data with many dimensions as machine learning or spatial database systems. When we increase the dimensions of

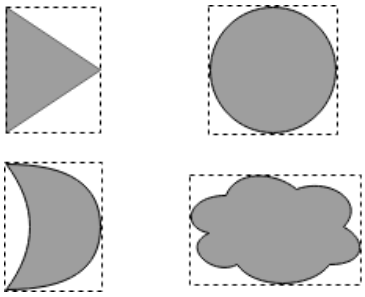


Figure 3.1: Four shapes in $\mathbb{R}^2$ with their respective MBBs (dashed lines).

the data the complexity also increases. The data is also more sparsely distributed. This have resulted in a split in focus among the indexing algorithms where some only focus on lower dimensional data while others specifically focus on solving the problem with indexes for high dimensional data.

Some techniques are created for high dimensional spatial data as locality-sensitive hashing. This is a technique where we use a hash function to place the spatial objects in 'buckets'. Points in a bucket are spatially close to each other. This technique has for example proven to be very useful for k-nearest neighbor queries for high dimensions. The drawback with this method is that it only calculates the approximate result since the hash function is not guaranteed to place all objects perfectly.

To limit the scope of this thesis we will focus on lower dimensional data.

3.2 BVH and R-trees

Bounding Volume Hierarchies (BVH) are commonly used when indexing spatial objects. BVH is a broad term used on algorithms that use hierarchical bounding volumes to structure the spatial objects. The bounding shape can be any shape as a box or a sphere. This structuring makes search and manipulation of the objects much faster.

One specific type of BVH structure is the R-tree. The R-tree is a spatial index which structures MBBs of objects in a hierarchical tree. Internal nodes in the R-tree contain a set of entries. R-trees became a popular subclass of the BVH tree because of its small memory footprint and efficient ways of doing intersection and distance computations. Each entry is represented by a MBB and a pointer to a child node. The child node could be another internal node or a leaf node which contains an object. All MBBs of descendant nodes are enclosed in the MBBs of their ancestor nodes. Some variants of BVH indexes use MBBs that are not axis aligned. Reasons for doing this is to better fit surround spatial object to reduce the chance of false positives. These boxes are called Object Oriented Box (OBB). R-trees though use whats called Axis Aligned Bounding Boxes (AABB). Through the reset of this when MBB is used it is equivalent to an AABB unless specified otherwise.

The formal definitions are written below as well as an example of a R-tree in Figure 3.2.

Definition 4. (R-tree entry) An entry E can be an internal node as defined in Definition 7 or a leaf node as defined in Definition 6.

Definition 5. (R-tree internal node) An internal node N_I contains a set of child nodes where each element of the set contains a pointer to another entry defined in Definition 4 and that entries MBB.

Definition 6. (R-tree leaf node) A leaf node N_L must have exactly one parent node and no child nodes. It holds a pointer to a spatial object Definition 2.

Definition 7. (MBB internal node) If we have an internal node N_I, the MBB of N_I will enclose all MBBs of descendant nodes: MBB(N_I) = $MBB(\cup_{E \in N_i} MBB(E))$

Definition 8. (Axis Aligned Bounding Box) Has the same enclosing properties as a MBB in Definition 3. We will define the axis aligned box B = (**a**, **b**) using two points $\mathbf{a}, \mathbf{b} \in \mathbb{R}^d$:

$$\forall i \in \{1...d\} | \mathbf{a}^{[i]} \leq \mathbf{b}^{[i]}$$

$$B = \{\mathbf{p} \in \mathbb{R}^d | \forall i \in \{1...d\} : \mathbf{a}^{[i]} \leq \mathbf{p}^{[i]} \leq \mathbf{b}^{[i]}\}$$

where $\mathbf{x}^{[i]}$ represent the i_{th} coordinate in the point vector.

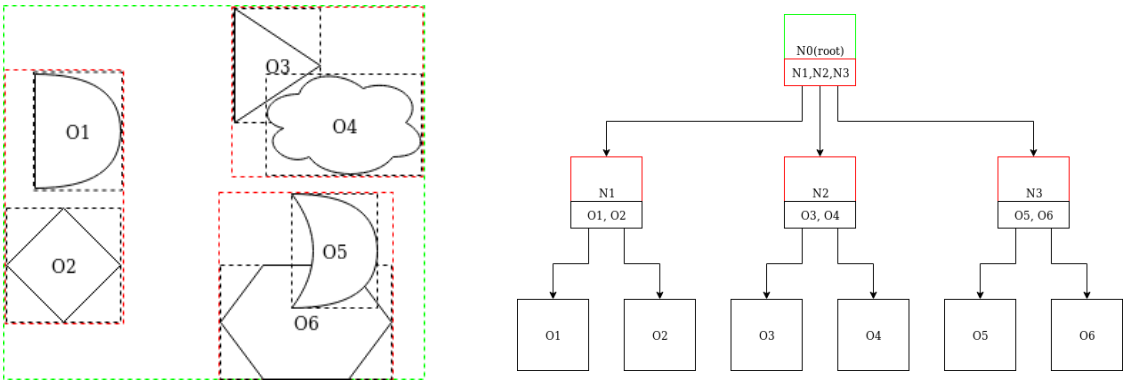


Figure 3.2: (Left) The spatial objects with their MBBs. Root level is green, the internal nodes are red and the leaf nodes are black.(Right) The R-tree structure corresponding to the spatial objects on the left figure.

The benefit of structuring the spatial objects in R-trees is the improved search performance. When searching for objects in a traditional range search the search starts at the root node, and checks if the search query intersects with any of its entries. This will be referred to as a node scan. If an entry does intersect the search continues scanning the node it intersected with. If an entry does not intersect we can prune the whole branch with all the descendant nodes. The pruning of these branches in addition to the fast comparison of MBBs is what makes the search efficient. This process continues digging down the tree in a recursive manner until leaf nodes are reached. If the search intersects with the MBB of the leaf node a more detailed check is needed to see if they actually intersect.

It is then clear that when building the R-tree we want as few overlapping boxes as possible. This is because overlapping MBBs can potentially lead to unnecessary node scans. This is also why we want to use minimum bounding boxes and not just any box representing the spatial objects. Imagine an R-tree with all bounding boxes for all levels covering the same space. For each node scan we would not gain any information, and all objects would be potentially intersecting meaning we would have to do a thorough intersection check for all objects in the tree.

We also want to find a balance for how many children a node can have. Too few will lead to deep trees, while too many will lead to shallow trees with many entries per node that have to be searched. Since R-trees are balanced there is a minimum and maximum amount of children in

internal nodes(except for root) N bounded by $m \leq |N| \leq M$, where the minimum m is $m \leq \frac{M}{2}$ and M is the maximum. If a node overflows it is split and the change is propagated up the tree. If it reaches the root node the root is split and a new level is made. This also ensures that all leaf nodes are at the same level.

There are many variants of the R-tree trying to improve upon different aspects. The R*-tree and the Hilbert R-tree are examples of this. They both try to improve different things in different ways. The R*-tree is focusing on improving the quality of the tree. This results in a bit worse construction time, but much better query performance. The main focus the R*-tree has is to minimize overlap between the MBBs and also their coverage. Coverage in this setting is the extent of the MBBs and is defined in Definition 9. Having as small coverage as possible is beneficial since having small MBBs in the tree will result in more pruning since the search have less chance of needing to search it. This is also the reason we want to have no overlap between MBBs since the overlap prevents us from prune.

Definition 9. (AABB coverage) The coverage of an AABB B = $(\mathbf{a}, \mathbf{b})$ as defined in Definition 8 is defined as:

$$Coverage(B) = \sum_{i=1}^{d} (\mathbf{b}^{[\mathbf{i}]} - \mathbf{a}^{[i]})$$

The disadvantage of R-trees are the cost of building and maintaining them. There are many algorithms created for building. Some focus on efficient creation, while others focus on a high quality tree that allows for fast searching. The quality of the R-tree is mainly determined by the amounts of MBB hits we get when searching for a spatial object. Many hits is an indicator that there is a high degree of overlapping with many unnecessary node scans. Since the tree is balanced the cost of inserting a node can also be high since splitting nodes can propagate changes upwards in the tree and cause expensive structural changes. The branching nature of the R-tree is also a huge problem for GPU. Since the GPU is running threads in warps that always execute the same instructions branching may create thread divergence which leaves much of the GPU hardware idle. Irregular memory access patterns is also a problem since the GPU memory is best utilized for consecutive memory instructions the tree structure is a problem.

3.3 Uniform grid

The uniform grid index also called epsilon grid has been widely used in particle physics simulations on the GPU. Much research on efficient spatial indexes on the GPU have been done through fields as particle physics and computer graphics. NVIDIA have used this index themself for particle simulation [18]. In particle physics we often want to simulate a finite number of particles to represent a fluid as for example water. All these particles can interact with each other. One way of doing this is through Smoothed Particle Hydrodynamics (SPH) [19] where the effect of forces interacting between particles drops of when the distance between them increases. It was originally developed for use in astrophysics but have been adopted by many other fields.

Both in [20] and [9] they use this structure to achieve great results on distance join queries. In [20] they use the grid index structure for spatio-textual similarity joins while in [9] it is used for spatial self-join on GPU. They both show significantly better performance than the previous state of the art methods which is promising.

The main idea is to make a finite space containing all spatial objects, and then dividing this space into a grid structure. Each cell in the grid have size of ε in all dimensions. Epsilon can be compared to the distance threshold in our top-k SDJ query. Since each cell have size ε we only need to compare the objects in a particular cell with all surrounding neighboring cells.

The uniform grid structure scales up with dimensions so for 2 dimensions we would only need to check 3^2-1 cells while for 3 dimensions it would be 3^3-1 cells. This dramatically reduces the number of points needed to be checked to $\mathcal{O}(n \times m)$, where n is the number of points and m is the average points in a cell. One problem with this structure is that it is unbalanced. When there are spaces with high concentration and most objects are placed in a few neighboring cells then the effect of the structure is not that great. The structure works best if the objects are uniformly distributed. This may not problem for the top-k SDJ query as the points in the neighboring cells would need to be tested anyways. By hashing the cells we can also preserve locality when processing the grid which will help with balancing the work in the SMs since points that are processed that lay in the same cell will have the same neighboring cells which will result in better cache utilization.

To implement this we need to first map the points to a grid, and then find an order that is best suited for traversing this grid that at the same time preserves the spatial properties when stored in GPU memory. This algorithm is a bit complicated so instead of describing the construction with pseudo-code the process will be described in detail below.

Mapping objects to the grid

First we need to map the spatial objects to the grid. This can easily be done by dividing the coordinates by the cell size ε for all dimensions and taking the floor of each dimension: $GridPos(\mathbf{p}) = \lfloor \mathbf{p} \times \varepsilon^{-1} \rfloor$, where $\mathbf{p}$ is the vector representing a spatial object. Figure 3.5 shows the result after the first step. We now have one array containing all the spatial objects and one array of the same size containing the cell ids of all the objects.

Cell id hashing and sorting

The cell id is responsible for representing the cell in one dimension. The result of the GridPos function in the mapping phase actually gives a vector of the same dimension as the points. To represent this grid in memory as a one dimensional array we need to hash the cell values. This can be done in many different ways.

One way to hash the cells is just a linear hash function. This will represent the cells the same way as in Figure 3.5. We then need to sort the array containing all the spatial objects based on these hash values. This will make sure that cells are processed in a linear order which is beneficial for thread divergence. This is because when warps can process points in the same cell each object have the same number of points they need to compare with.

A downside to this method is memory layout. If we consider the row-wise linear hash in Figure 3.5 we get a good row-wise structure, but if we want to compare objects with objects on another row the memory locations are far apart. This is bad for caching.

One ordering that helps with this is z-ordering. The idea is to order the cells in a one dimensional way that keeps in mind their multidimensional positions. To calculate the z-order cell id we first need to convert the cells dimensional values to binary. This means that cell (2,1) in decimal is replaced by (010, 001). To calculate the z-order:

$$z = f(x,y) = (x_0, y_0, x_1, y_1 \ldots x_n, y_n)_2 \tag{3.1}$$

Example of (3.1): $z = f(010,001) = (001001)_2 = 9$. If we do this for all cells and sort the points based on the hash values we can traverse the points in z-order. Figure 3.4 shows both z-ordering and linear order hashing for a grid of 4x4 cells.

There is still one problem with the z-order curve. There are still some long jumps in space. To solve this we may want to use a Hilbert curve. Hilbert curves never have longer jumps than one cell while still preserving the multidimensional locality. The only disadvantage compared to z-ordering is that Hilbert curves can be more difficult to calculate while z-order curves are very simple. Figure 3.4 shows an example of all the different curves.

The Hilbert space filling curve was introduced by David Hilbert in [21]. The fact that the points on a continuous curve can be mapped uniquely to the points of a square is pointed out.

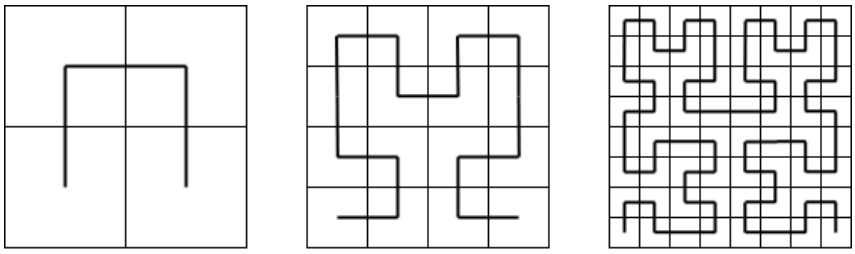


Figure 3.3: The first, second and third order Hilbert curve respectively from left to right.

Figure 3.3 shows how the curve is constructed. The curve starts out with a predefined curve covering 4 quadrants of a square. This is called a first order Hilbert curve.

The second order curve is generated by copying the first order curve four times to fill the four quadrants of a new square and rotating the bottom two before two curves to connect them.

Rotation and connection of the pattern at the step before, we get a Hilbert curve of order 3.

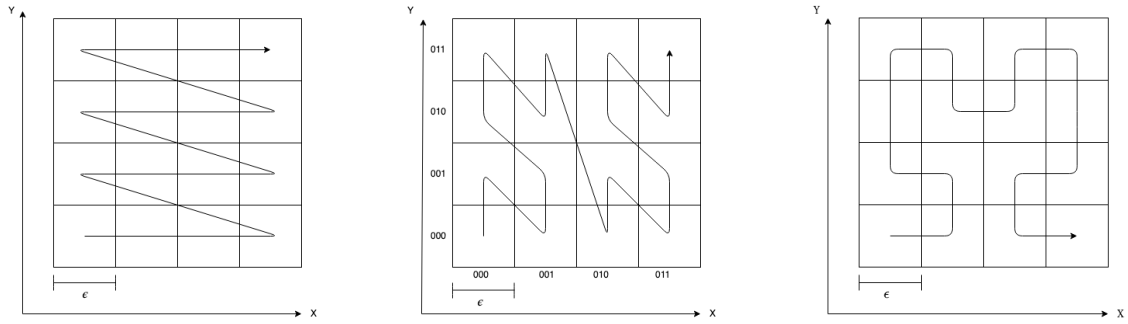


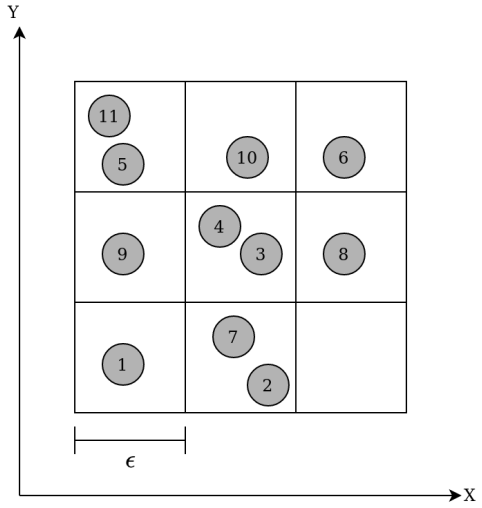
Figure 3.4: (Left) A linear row-wise ordering. (Middle) A z-order ordering. The the binary numbers represent the cell id positions in binary. (Right) A Hilbert curve ordering.

Finding cell start and end

Since our task is to compare objects of neighboring cells we need to know where cells start and end. We only have a one dimensional array so we need to store the cell start and end positions somewhere else. Since the cell id hash values are ordered we can find cell start and end values by traversing the array and detecting where the hash value changes. If it does we know that we have a new cell.

Comparing cells

Now we have built our grid index structure. The only thing left is to compare the cells. To do this we can launch a kernel with one thread per object. Each thread compares the object it is given with all the objects of its neighboring cells. Since we have taken thread divergence and memory access patterns into account we can expect the GPU utilization to be high. This structure also fits the top-k SDJ problem perfectly as the ε distance also is utilized.



Cell id	Object id
0	1
1	2
4	3
4	4
6	5
8	6
2	7
5	8
3	9
7	10
6	11

Figure 3.5: (Left) The spatial objects divided into the grid structure.The bottom left cell has id 0 while the top right cell has id 8.(Right) The list of the spatial objects with their respective cell id. The cell id is retrieved by a linear hash of the cells position.

3.4 Choosing a spatial index for the top-k SDJ problem

The nature of the GPU execution model makes spatial indexing challenging. The reason is that the hierarchical and branching nature of the structures actually makes them slower on the GPU if implemented as on the CPU. One of the main reasons for this is that all threads in a warp execute the same instruction making thread divergence (Section 2.3.2) a huge problem. This also makes memory access patterns not utilize the cache of the GPU to its full potential. Since global memory access is so slow this has a substantial performance impact.

There is research done on implementing these R-tree on GPU in [5], [6], [7]. They achieve respectable results but the nature of the spatial distance join problem makes the epsilon grid a better choice. This does not mean that the other structure does not have a potential to be a good choice for for example box data. But since we are trying to solve a problem for point data with the distance predicate the uniform grid structure seems like the best choice.

The conclusion that grid structures are suited for GPU also aligns with previous research [8], [9] and [11]. This including the previous findings by [20] which shows that the grid structure even out performed a R-tree structure on CPU for a problem very similar to our spatial distance join problem. This indicates that the uniform grid structure is the best choice for our top-k SDJ algorithm.

Chapter 4

Spatial join algorithms on GPU

In this chapter methods for spatial join on GPU will be discussed. The methods will be implemented based on the descriptions in the papers and benchmarked to find their strengths and weaknesses. The benchmarking setup can be found in Chapter 6.

The aim of this chapter is to give a solid understanding of which methods and techniques are suited for GPU. Since there may be relevant techniques in different types of spatial join we have chosen not to limit our investigation to only SDJ join on point data.

We will first look at a naive brute-force nested loop approach in Section 4.1 and see how it compares to more complex methods. The second method in Section 4.2 called MLG-join will be a spatial intersection join between boxes. The third method will be an intersection join method called GCMF in Section 4.3. It is made for polygons where the focus is on the MBB filtering algorithm. The last method in Section 4.4 will be a spatial self join method for point data. The idea with investigating such a range of problems is that since the field is new and there are no other top-k spatial distance join algorithms on GPU it may be important to get a broad understanding of the filed to pick up ideas that would otherwise be missed.

The things we learn in this chapter will be used in the development of the new top-k SDJ algorithm.

4.1 Nested-loop join on GPU

The traditional nested loop join is relatively slow. The complexity of $\mathcal{O}(|R| \times |S|)$ makes even reasonable problem sizes of 10^6 element datasets very slow to compute since it would require 10^{12} that is one trillion comparisons. Processors today use gigahertz clocks, so even if we had managed to make hardware that could compare two objects in only one clock cycle we would still need 1000 seconds for this relatively small problem size.

This clearly shows the importance of algorithms that can reduce the search space drastically. Instead of reducing the search space nested-loop join on GPU utilizes the parallel nature of the problem to its full potential.

The idea is that since thread divergence and bad memory access patterns leaves some of the GPU hardware unused it may be better to do the spatial join as simple as possible where the access patterns allow for coalesced contiguous memory requests. Since all computations are individual it is trivial to parallelize the algorithm. Each thread on the GPU represents the objects in dataset R. Each of these threads will loop over all the objects in S and check if the join predicate holds. If it does it is appended to the result set.

Algorithm 2: The Naive GPU algorithm

```
   input  : R, S
   output: Result
1  index ⟵ (blockIdx.x * blockDim.x) + threadIdx.x ;
2  if index > |R| then
3  |   return ;
4  foreach s ∈ S do
5  |   if predicate(R[index], s) then
6  |   |   resultIndex ← atomicAdd(resultIndex, 1) ;
7  |   |   Result[resultIndex] ← Result ∩ (r, s)
8  end
```

Figure 4.1 shows how the runtime is effected by different densities and dimensions. Different densities have close to no effect on the runtime since there are no index structure that can be effected by the density. There is no difference for the brute-force algorithm checking distances in dense data compared to more sparse data. The difference we see is from the increased result set size.

Changing the dimensionality of the data will on the other hand effect the runtime since more data is transferred and the intersection check complexity scales with dimensions. This method will act as a baseline to the other methods we will implement in this chapter.

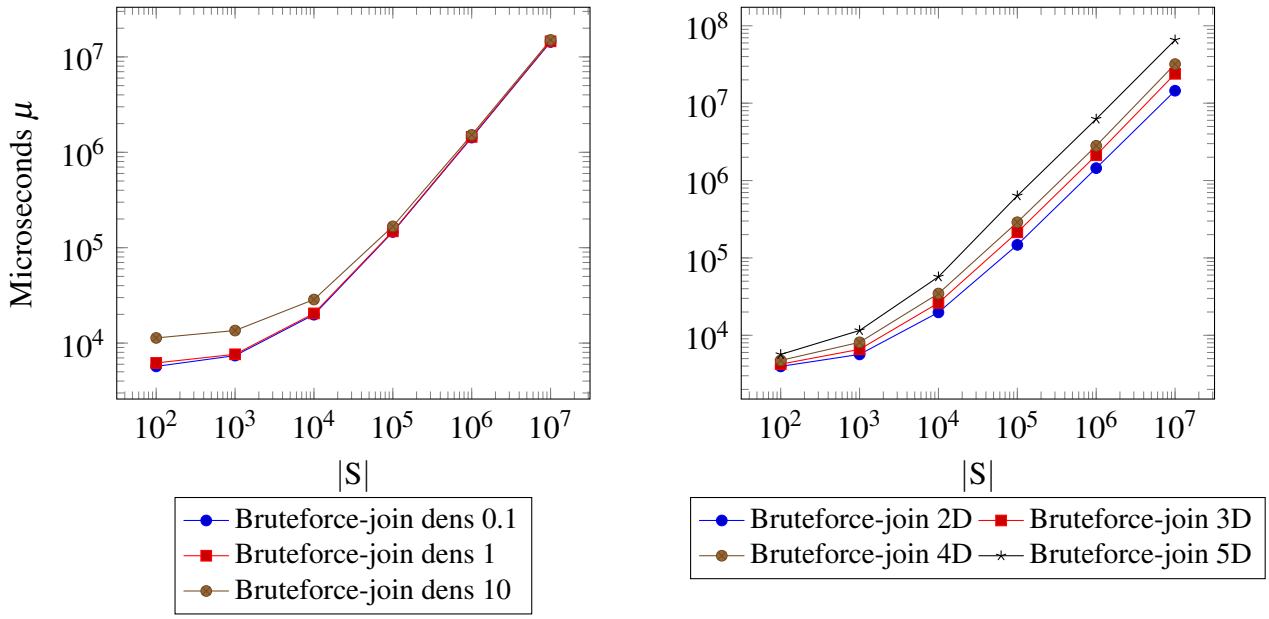


Figure 4.1: (Left) The runtime of the brute-force nested loop algorithm for different densities. (Right) The runtime of the same brute-force algorithm for different dimensions.

4.2 Multi Layered Grid join

The Multi-Layered Grid (MLG) join algorithm [8] introduced by Ward et. al is a spatial intersection join algorithm for boxes designed for GPU. It was intended to solve the problem of

real-time continuous intersection joins over large sets of moving objects. Since it is a continuous intersection join algorithm it is designed so that updates to the index is efficient. Since we are only concerned with static joins we can ignore the update steps and only focus on the initial join process.

4.2.1 The MLG-join process

The first step in MLG-join is to build a multi-layered grid structure. The idea is to partition the R dataset into N equal partitions, and make a grid out of each partition. This is why its called multi-layered. In contrast to many other spatial algorithms using partitions this algorithm does not base its partitions on the spatial information of the objects. Instead the partitions are logical and consecutive meaning that each partition is just consecutive partition laying in memory. No need to move or transform any data.

This step is best illustrated with a simple example. Lets say we have two datasets of boxes R = { r1, r2, r3, r4 } and S = { s1, s2 }, and we want to find which boxes intersect. Figure 4.2 illustrates the objects drawn on top of the grid. We choose to make two partitions of R called P_1 = { r1, r2 } and P_2 = { r3, r4 }. We then project each object onto the grid. If at least one object is projected onto a cell it writes 1 and if no objects are projected onto it the value is 0. The final index is the combination off all partitions projected onto the grid. Figure 4.3 illustrates the final grid index.

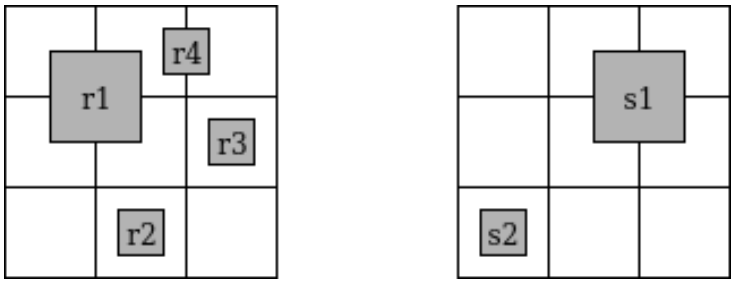


Figure 4.2: (Left) All objects in R drawn on top of the grid. (Right) The objects in S drawn on top of the grid.

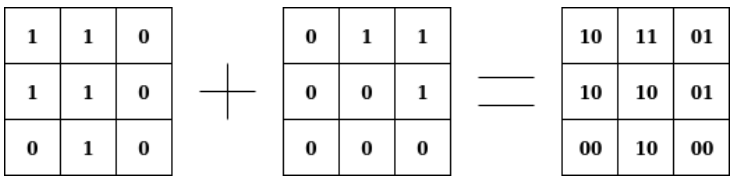


Figure 4.3: The two first matrices are the result of projecting the objects in P_1 and P_2 onto the grid respectively. The resulting Grid index is the combination of all the partitions, where each bit position represents the value of that partition.

Some of the main reasons the authors give for using a multi-layered grid is that they need no atomics or locking mechanisms to update the index. In addition, the index does not use dynamic memory allocation which is important for GPU performance.

The next step is to reduce the search space. We want to probe the objects in S against the grid-index we have made in the previous step and see which partitions that have objects that potentially intersects.

Object is S	Bit string
s1	11
s2	00

Partition S	Object form S
p_1	s1
p_2	s1

Table 4.1: (Left) The resulting bit string list of the probing step. (Right) The resulting object lists for each partition after stream preparation.

The result is a bit string for every object in S, where a bit represents whether or not the object from S intersects at least one object in a partition of set R. Zero in an bit string means the object in S does not intersect any object in the partition corresponding to that bit. Table 4.1 show the result of this step.

The stream preparation step prepares the objects in S for the actual intersection checks in the next step. This step can be thought of as a transposing step where instead of having a bit string for each object in S we make a list for each partition where the elements of the list are the objects that had a 1 at that partition in its bit string from the probing step. The result is in Table 4.1.

This stream preparation step is problematic for memory performance as the size increases with both the number of partitions and the size of S. Finding a way to not materialize the grid structure for the top-k SDJ algorithm may be important to not make memory capacity a limiting factor as the GPU main memory is small.

The only step left now is finding the intersecting objects in the reduced search space. To do this we need to iterate through all partitions and for all the objects from R in that partition we check if they intersect with any of the objects in the stream that corresponds with that partition.

4.2.2 Performance and analysis

Figure 4.4 show how the cell size and partition parameter affects runtime performance for different problem sizes. The results confirm the findings in the original paper that the cell size is optimal when it is about the same size as the box objects in R. In this case the objects were rectangles with dimensions ca 0.003. That explains why the cell size of 0.001 and 0.01 perform best.

MLG-join perform well for large problem sizes but struggles on lower sizes. This problem will occur when the number of objects is significantly smaller than the maximum number of threads in a block for the kernel that will create the grid. This will result in under-use of the hardware. This penalty can be minimized or completely avoided through parameter selection. For example when selecting an appropriate cell size the performance is good also for small problem sizes.

For number of partitions the graphs show that more partitions is better. What limits us from having more partitions is the memory footprint of the grid index when increasing the number of partitions.

Figure 4.5 gives an overview of how the density of the data impact performance. Since the problem sizes are the same for all the problems the density is determined by the size of the box objects. When the object density increases the runtime will also increase as both the number of boxes that overlap is increasing but also the number of cells that have a potential overlapping box in the grid. This is a downside of using a grid index compared to for example a brute-force method that is not affected by the density as much. Figure 4.5 shows how the MLG-join method compares to the brute-force method. As we can see the brute-force method is much slower, but

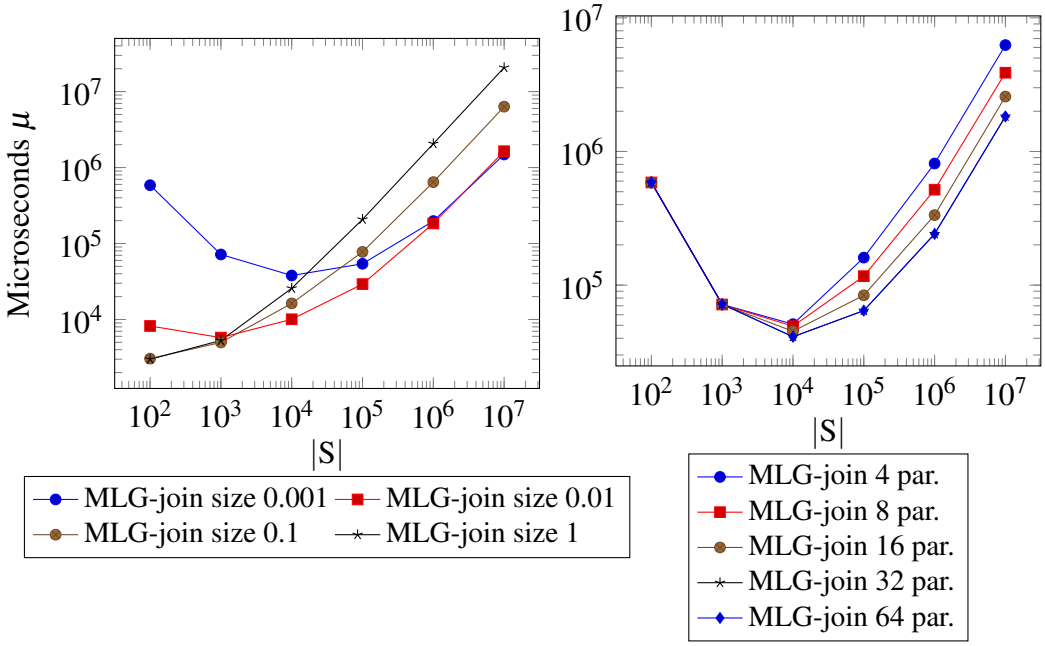


Figure 4.4: (Left) The cell size parameter effects the runtime of MLG-join. (Right) Show how the number of partitions effect the runtime of MLG-join

not as effected by the density as MLG-join. If we were to increase the density even more the index would render useless ass all boxes would cover the same space which would result in the MLG-join method doing more work than brute-force.

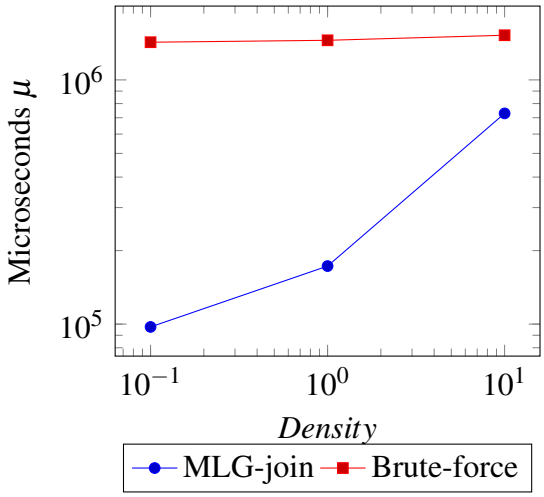


Figure 4.5: The runtime of MLG-join and brute-force for different dataset densities.

Even though there is a problem with using an index the positive aspects outweigh the negative. Figure 4.6 show how the brute-force algorithm compares to the MLG-join algorithm on different problem sizes. The graph clearly show that the MLG-join is a better option for all but the smallest problem sizes.

The memory usage is also effected when changing the parameters. Figure 4.7 show that the memory usage increases drastically for relatively small problem sizes. This clearly show the dis-advantage of materializing an index structure. Both decreasing the cell size and increasing

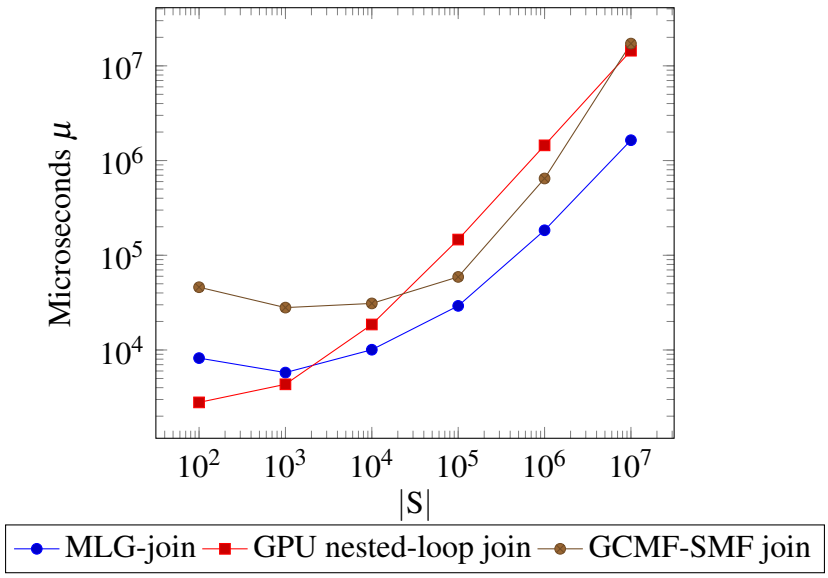


Figure 4.6: The runtime of MLG-join compared to an optimized nested loop join and the SMF step of the GCMF algorithm. |R| = 10^5 and a uniform distribution with density 1 is used.

the number of partitions heavily increase the memory usage and therefor limit the usage on GPU because of the limited device memory size. Finding a method to not materialize the grid index for the top-k SDJ problem while still having efficient look-up could be important.

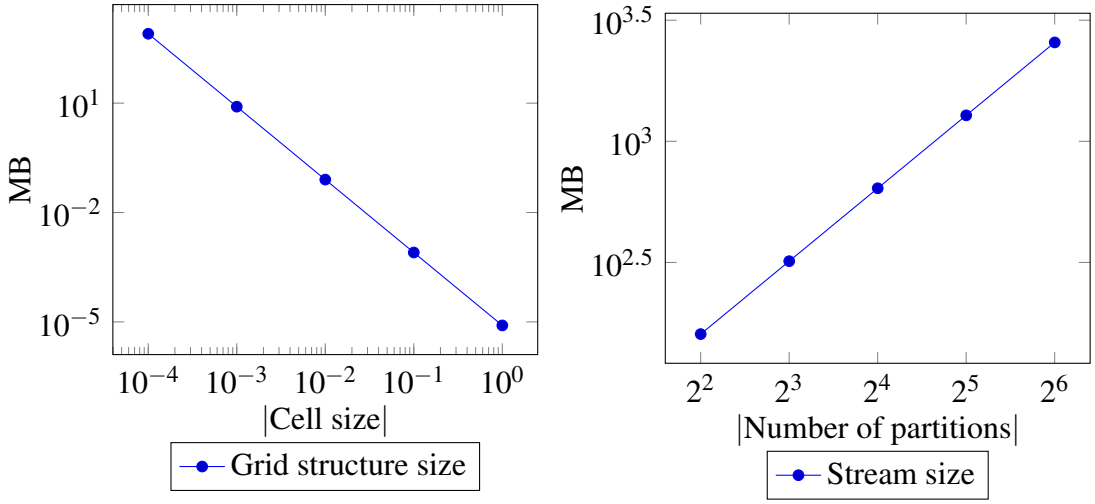


Figure 4.7: (Left) The memory usage of MLG-join with when varying cell size with problem size $|R|$ = 10^5 and $|S|$ = 10^7. (Right) The memory usage of MLG-join stream size when varying number of partitions for the same problem size.

The 3D surface plot in Figure 4.8 show how memory is affected by the input dataset sizes. The important points here is that increasing $|S|$ significantly affects the memory usage of MLG-join. This is mainly because both the probe structure and the stream structure increase based on $|S|$ and not $|R|$. As noted in Figure 4.8 the total memory usage of MLG-join is significantly affected by the stream structure size. Finding a way to reduce this size would be beneficial as the limit of GPU global memory is quickly reached when using this structure. It will also be

important to try to balance the memory footprint of increasing R and S so that the imbalance wont effect how large problems we can execute.

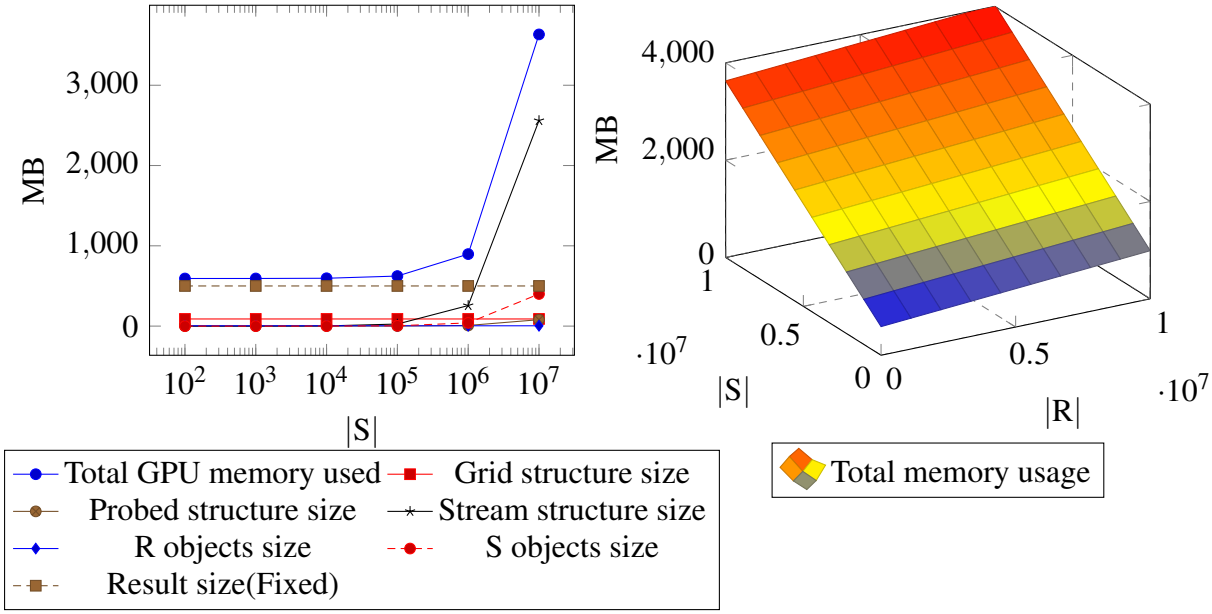


Figure 4.8: (Left) The memory usage of MLG-join with cells size 0.0003 and 64 partitions over the problem size $|R| = 10^5$ and $|S| = 10^2$ to 10^7. (Right) The MLG-join memory usage for different input size combinations.

4.3 GCMF

The GCMF spatial join method [10] is based on the filter and refinement structure used in many other spatial join systems Section 2.1.1. In contrast to many others this method does not use an index structure as grids or R-trees. It is designed for intersection tests between polygons so the main focus of the article and the algorithm is on the filtering step. Figure 4.9 shows an overview of the system. The focus in this section will be on the filtering process for two reasons. Firstly the refinement step of polygon intersections is a complex field that could have multiple thesis on itself on how to do it efficiently on GPU. Secondly the refinement step is not really focused in the paper itself so the new ideas introduced in this paper mainly come from the filtering step.

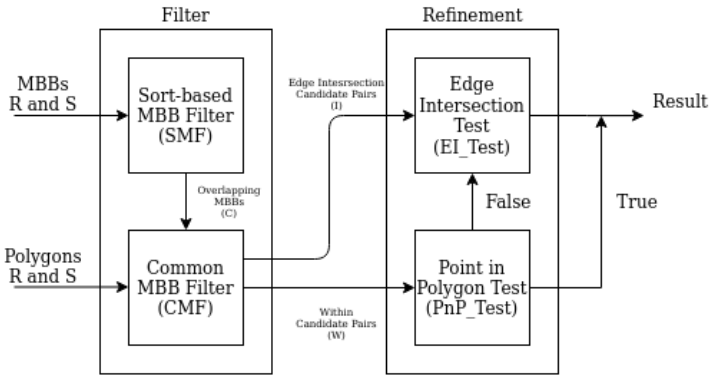


Figure 4.9: An overview of the GCMF spatial join system. The original figure can be found in [10].

Algorithm 3: The SFM algorithm

input : $MBB(R), MBB(S)$
output: C

```
1  X = {x_i | x_i ∈ x − coordinates of R ∩ S} ;
2  (sortIndex, RankIndex) = RadixSort(X) ;
3  foreach GPU-block i, 0 < i <(m + n) do
4      foreach x_{j0}, x_{i0} < x_{j0} < x_{i1} do
5          if (y_{i0}, y_{i1}) insersect (y_{j0}, y_{j1}) then
6              if MBB_j ∈ S then
7                  Add (i,j) to C ;
8              else
9                  Add (j,i) to C ;
10     end
11 end
```

4.3.1 SMF

The first filtering algorithm is Sort-based MBB Filter (SMF). It does not use an index as many other algorithms but instead bases itself on sorting for some inherent structure. The algorithm starts out by finding the MBBs of the two sets of polygons R and S such that $p \in R,S$ have a MBB MBB_p. The next step is to find pairs of overlapping MBBs. The result is a candidate set C where $C = \{(i, j)|i \in R, j \in S\}$. The pseudo algorithm can be found in Algorithm 3.

The first step is to make an array of all the x-values of the MBBs in R and S. The array is labeled X. The next step is to make two new arrays called SortedIndex and RankedIndex. SortedIndex is made from taking the indexes of the elements in X and sorting them based on their X values. This means that when you index SortedIndex[i] you get the index of the i-th smallest element in X. To keep track of the new positions of MBBs in the new sorted array we use the RankedIndex array. It will give the index of where the MBBs moved to in the sortedIndex. Example:

$$MBBs = \{(1,6),(2,4),(3,9),(7,12),(10,13)\}$$
$$X = \{1,6,2,4,3,9,7,12,10,13\}$$
$$X_Sorted = \{1,2,3,4,6,7,9,10,12,13\}$$

$$SortedIndex = \{0,2,4,3,1,6,5,8,7,9\}$$

$$RankedIndex = \{0,4,1,3,2,6,5,8,7,9\}$$

Algorithm 3 uses these indexes on line 4 to efficiently find the MBBs that have an x value between the two x values of the MBR that is checked. This will give us a set of MBBs that overlap with the MBB we are checking in the x dimension. This set of MBBs are then checked with the checked MBB if they overlap in the y dimension. If they are we can add a new pair to C witch is the result set from this step.

4.3.2 Performance and analysis

The other steps in GCMF are one more filtering step called CMF and a final refinement step which do the actual polygon intersections. Since the algorithm was supposed to handle polygons it can afford to have multiple relatively expensive filtering steps since the polygon intersection test are so complex. For our top-k SDJ problem we can think of the points with distance checks as the filtering and the top-k part as the refinement. Compared to polygon intersections the ε distance checks are much less demanding. Finding a method that like GCMF filters out as much as possible from the expensive refinement step could be a good idea to use for our top-k SDJ algorithm.

One improvement the GCMF method has over the MLG-join method is the memory usage. Since it is not using an index the space problem does not become as big of a problem as for MLG-join. The GCMF method is therefor able to run larger problem sizes not limited by the GPU device space. Figure 4.10 shows the total memory usage.

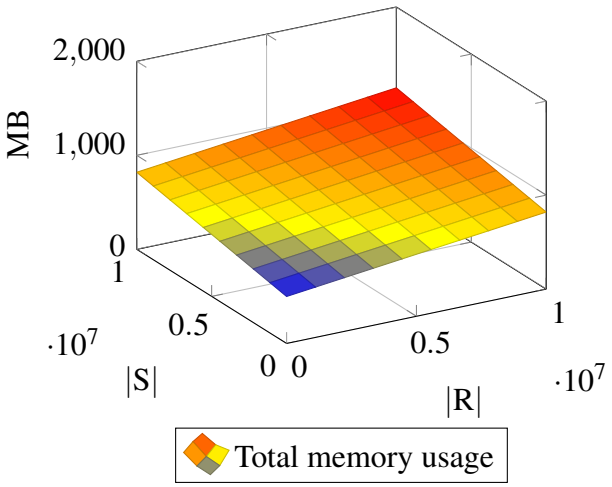


Figure 4.10: The memory usage for different input size combinations.

Like MLG-join GCMF also struggles with density. As Figure 4.11 shows the runtime is significantly effected by increasing the density. This is because the more MBBs needs to be checked for intersection when there are more MBBs that have overlapping x-axis.

The comparison of the other methods in Figure 4.6 shows that the GCMF-SMF algorithm struggles to perform better than the brute-force method. MLG-join shows significantly better performance likely because more potential intersections are filtered out. This teaches us that even though memory usage is better controlled the benefit of having an index may outweigh the negatives.

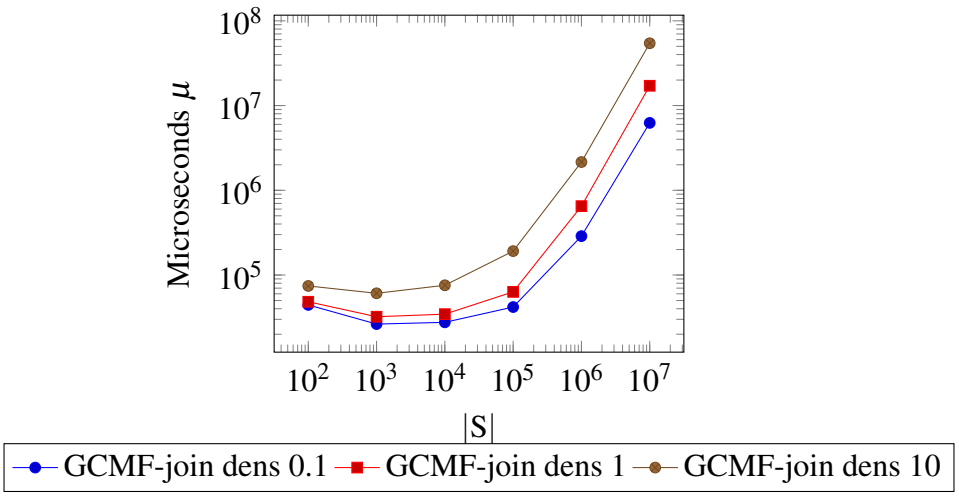


Figure 4.11: The runtime of GCMF over datasets with different densities.

4.4 Self-join on point data

Gowanlock and Karsin presents an efficient self-join algorithm for spatial distance join on point data in [9]. The algorithm performs well compared to state of the art algorithms on multi-core CPU platforms. The method utilizes a grid structure that is optimized for GPU and a batching scheme. The algorithm is made to handle distance comparisons similar to our top-k SDJ algorithm.

The grid index is very similar to the uniform grid structure discussed in Section 3.3. The difference is that instead of needing to materialize all cells, this grid only allocates space for the grid cells that have at least one object in them. This method significantly reduces the space allocated for the index. The original paper [9] explains well how this is done. The method checks for empty cells and only sends cells with points in them to the GPU. Masks was also used to not check empty cells on the GPU side. Since grids are already explained in Chapter 3 this section will not focus on this again. Using this knowledge the new top-k SDJ algorithm may benefit from not materializing the grid.

4.4.1 Batching

Since the GPU memory space is heavily limited it is important to handle result sizes that overflows this limit. Batching is a technique that can be used to overcome this issue [9], [22]. By using a batching scheme we can not only prevent memory overflows but we can also reduce memory usage on the GPU and use memory streams to do data copying and processing simultaneously.

The idea of the batching scheme is to use result size estimation to determine how many batches are needed. If we knew the result size exactly we could use the simple equation:

$$n_b = \frac{a_b}{b_b} \tag{4.1}$$

,where a_b is estimated total result size and b_b is allocated result size space on the GPU.

This is unfortunately not the case for two reasons. The result size will vary between batches and the estimated result is only and estimate. To overcome this issue we need to make an

overestimation. In [22] they introduce the overestimation factor α and the equation is changed to:

$$nb = \left\lceil \frac{(1+\alpha) * a_b}{b_b} \right\rceil \tag{4.2}$$

The estimated result size a_b is must be calculated based on the query. The method used in [22] is to take a uniform sample of points from from the dataset and run the query on those points but only save the result size. This sample run will be much faster than the actual query. This assumes that every batch will produce approximately the same result size. To achieve this you may need to customize the query algorithm. a_b is calculated by:

$$a_b = eb(f) * \frac{1}{f} \tag{4.3}$$

, where e_b is the estimated result size of a fraction of size f.

Batching allows kernels to run simultaneously while copying data from the host to the device. This is called streams in CUDA. To enable stream data transfers we need to pin the data on the host side. One positive effect of this is that data transfers can be faster when the memory is pinned. This is because data allocations are pageable by default. Since the GPU cannot access data directly from pageable host memory, so when we want to transfer data from pagable memory to the device the CUDA driver must first allocate a temporal page locked space before we can transfer the data to the device.

If we pin the memory allocated on the host from the start this step is not needed allowing for much higher memory transfer speeds. This will save time for two reasons. The first is that kernel computation can be overlapped with memory transfers potentially solving the memory transfer time consumption often being a limiting factor with GPU algorithms. The second is that the memory transfer speeds are higher when we use pinned memory, so if the GPU is not able to overlap the transfers with kernel computation it at least have faster memory transfers.

Chapter 5

The new Top-k spatial distance join algorithm

The new top-k spatial distance join algorithm is build based on the knowledge we have gathered in the previous sections. The algorithm is constructed to run on GPUs with a massively parallel architecture. This chapter will describe the algorithm design in detail including complexity analysis. This will hopefully help with reproducibility and improvement suggestions from other researchers. The algorithm will then be benchmarked in Chapter 7.

5.1 Design overview

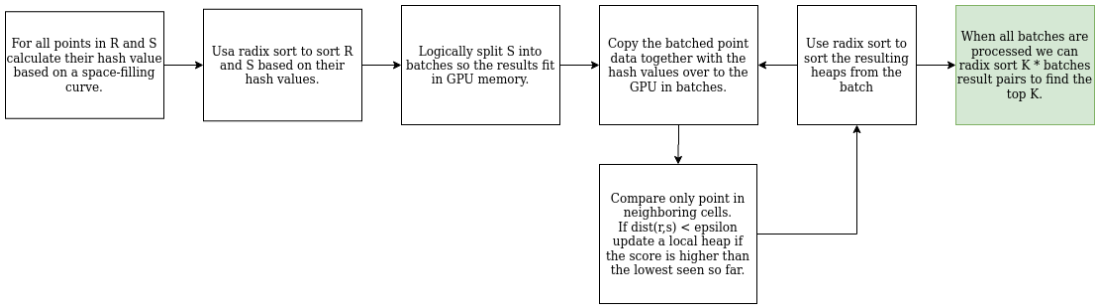


Figure 5.1: A diagram describing the new top-k SDJ algorithm.

The new top-k spatial join algorithm is made to solve the problem defined in Definitions 1. The spatial data will be point data in n dimensions. This section will cover the design choices that were maid when constructing the algorithm. To get an overview of the process Figure 5.1 shows the overall process. The upcoming sections will describe the design choices in more detail.

5.2 Indexing

The first step of a typical spatial join algorithm Section2.3 is to index the data in a spatial data structure for easy access to spatial objects based on a position query. This is also the first step in this algorithm. As shown in Chapter 3 there are many options to chose from where each

structure have their own advantages and disadvantages. The structure chosen for this algorithm is an epsilon grid.

There are three main reasons for picking this. The first is that epsilon grid is not a hierarchical structure. Many of the other traditional indexing structures as R-trees base their spatial reduction on reducing the options when traversing the structure hierarchical. For those algorithms the hierarchical division is good, but for GPUs this will lead to thread divergence leading to large amounts of hardware being idle. Since epsilon grids are direct access and not a hierarchical structure the epsilon grid seems to be a good fit for this problem. As discussed in Chapter 4 epsilon grids can also be used for spatial objects covering an area as rectangles or polygons. The difference is that instead of the cell width being the size of the epsilon of the query it is the mean size the objects cover in the space. This is demonstrated in Figure 4.2 and 4.3.

The second aspect that was important when choosing a structure is space complexity. Epsilon grids can potentially have space complexity of $\mathcal{O}(n^d)$ which can be especially bad when ε is small and the data is spread out on in a large space. To solve this problem we have come up with a technique that lets us not materialize the epsilon gird. The technique is based on doing binary search to find cell bounds in sorted arrays of hash values. We are able to do this since the spatial objects are sorted based on hash values that preserve spatial locality. Since binary serach is quite fast only requiring $\mathcal{O}(\log(|R|))$ steps at worst the cost of this compared to materializing the grid is justified. This also allows for arbitrary small ε values.

Choosing a key value sorting algorithm was quite simple. Since all hash values was unsigned integers of the same size a highly optimized parallel key value radix sort algorithm was chosen. The sorting algorithm is provided by the thrust library [23] for CUDA. Since the algorithm is using the GPU to perform the sort we don't need to transfer any data back to the host. The time consumption of this sorting step has shown to be insignificant compared to the total runtime of the algorithm

The third aspect that was important when choosing a spatial index was spatial locality. This is especially important for GPUs since we want objects that are located in the same space to be processed at the same time to avoid thread divergence. This is done by calculating the space-filling curve hash of all points in R and S and sorting the points based on this order. This has two advantages. The first is that we preserve the spatial locality of the objects while they are stored in a 1-d array. The second is that when processing this data we can find the start and end of a cell in just $\mathcal{O}(\log(n))$ time for all cells. We have tested 3 different space-filling curves linear, z-order and Hilbert and found that the Hilbert curve gave the best results.

5.3 Filtering and Refinement

The filtering step uses one GPU thread per point in S. First it calculates the cell the point belongs to in the epsilon grid. It then loops through all the neighboring cells. To find which points in R that are inside the neighboring cells we use two binary searches in the sorted hash values to find the upper and lower bound of hash values equal to the hash value of S's neighboring cells. If the difference between the upper and lower bound is 0 we now that there was no points from R in that cell and we can skip it. Only the neighboring of the point are checked. If we find a pair from R and S within epsilon distance we can update a thread local heap. The refinement step is to find the top-k form the reduced serach space of only K*batches potential pairs by sorting. We want to reduce the amount of points sent to this step as much as possible.

5.4 Finding the top-k

The local heap can potentially be stored in global memory, shared memory or register memory. Global memory is a bad choice since the latency is too high. This is both because the off chip memory is slow compared to registers and shared memory, but also because the access pattern would be impossible for the GPU to coalesced. Shared memory is a better pick. We can save the heap as an array in shared memory. The arrays can be striped so each thread in a warp can access a separate bank in the shared memory so we get parallel access. The disadvantage is that we have a very limited amount of shared memory. Since we need to allocate space for K * number of threads in a block items we quickly run out of space. For example if we have K = 64, threads in block = 64 and the size of each item stored set to 12B we get 49KB which is already close to the 64 KB available per SM on the Tesla P100(see Table 6.4)

During benchmarking the time used on this step has been insignificant compared to the filtering step. We can also add that other top-k methods were considered. The HRJN methods used in the top-k SDJ algorithms in [1] do not fit the parallel GPU architecture as they rely on an ordered procedure where all steps of the iterative process rely on the last. Another option that was considered was to calculate all pairs and their combined score within ε distance fist before using an efficient top-k key value algorithm as [14] based on radix sort on the GPU to find the top-k. The main problem with this method is that we would generate a huge amount of unnecessary results from the distance comparison step that would nearly all be unnessesary. It would also more costly batch size estimations as we would need to run a small sample of the problem to get an estimate the same way as described in Section 4.4.1. Calculating the local top-k values as described earlier reduces the result size of filtering potential significantly while also not needing a batch size estimation as the result size from all batches are known.

Also if the result size from the distance comparisons where too large to fit in memory we would need to split the top-k calculations into multiple steps. By our evaluation calculating the local top-k values solves all these problems and is therefor a much better option.

5.5 Batching

Batching for a top-k spatial join problem is actually significantly easier and faster than for normal spatial join since we already know the result size will be K from all kernels. There is no need for a result estimate which will save time. The user can specify how much device space that can be allocated for the result. From that parameter we can calculate how many objects from S we need in each batch. The only data that is not batched is the R set and its hash values. This means that if you can fit the size of R plus its hash values all problem sizes can be calculated. This virtually removes the bottleneck of small memory sizes on the device while still getting the benefits of the high speed device memory.

We were not able to find a method that could batch the R set as well. The problem is that we would need to find which points in R all the points in the S batch could potentially be within ε distance from. This is equivalent to finding the cells that surround the cells that the points in S cover. Doing this with a Hilbert grid was challenging and is left to future research.

5.6 Memory management

To utilize the GPU to its full potential it was important to ensure the design allowed for coalesced reads and writes. Since all objects in S are sorted based on the space filling curve we use one thread per object in S so each warp access consecutive objects in memory. Both R and S are Arrays of Structures (AoS). Other data structuring techniques as Structure of Arrays (SoA) was also considered. SoA is known for being good for GPUs since the arrays are made out of data types with guaranteed alignment compared to custom structs that do not necessarily align. They also allow for access of only the data that is requested and not the whole structure. During development both methods were tested and the performance of the two was very similar. Since the algorithm ended up needing all data in the structure when it was requested the performance difference was minimal. Also unaligned memory access does not have that much effect on transfer speeds as discussed in Section 2.3.2. Since the AoS approach was simpler to implement especially for the key value radix sort the AoS approach was chosen.

To ensure registers was used for the top-k heap we had to fulfill 3 requirements. The first is that the heap needs to have a constant size. For this problem it is K. The second is that arrays needs to be indexed based on constant quantities. This is ensured by the heap update process indexing the array in the interval between 0 and K which gives the compiler knowledge of which registers will be accessed. The third is that the compiler decides if it use too much register space. If it does the heap is placed in local memory instead. This will be affected by the K parameter. For reasonably small sizes of K the heap can be placed in register memory.

To ensure proper memory access for the shared memory heap we store the K arrays striped so that all threads store the heap in a single bank. This lets each thread in a warp access a separate shared memory bank which leads to full parallelization of both reads and writes to the heaps in shared memory.

5.7 Correctness

The correctness of the algorithm will now be discussed. The uniform grid index have cells of ε dimensions. When all objects in S are processed we check all neighboring cells that contain R objects. Since the radius around the points in S will never be larger than ε the neighboring cells are guaranteed to include the points in R which could be within ε distance. Since distance measures are reflective we do not need to check the other way around if R is withing distance of S. This guarantees that all pairs from now on are fulfilling the distance requirement. The next part is finding the top-k scores. We find local heaps for all object in S and add them to a partial result batch. The local top-k pairs have higher scores than all other pairs found for an object in S. Then these local heaps are added to the result batches. The batches are then also guaranteed to have the highest scores found so far. Then we sort the batches to find the top-k for each batch. This will guarantee that each batch have the top-k scores for all pairs in that batch. These top-k scores from each batch are then sorted a one last time to get the final top-k pairs. The same principle as for the batches is used for the final sort.

5.8 Algorithm and complexity

Algorithm 4: The new top-k SDJ algorithm in pseudo code

input : R, S, ε, γ, K, *ResultSize*
output: *Result*
hashValuesR $\longleftarrow$ hashKernel(R, ε);
hashValuesS $\longleftarrow$ hashKernel(S, ε);
keyValueRadixSort(hashValuesR, R);
keyValueRadixSort(hashValuesS, S);
batchSize $\longleftarrow$ getBatchSize(*ResultSize*);
partialResults $\longleftarrow$ $\emptyset$;
foreach $S_{batch} \in S$ **do**
    top-kFilterKernel(R, *hashValuesR*, S_{batch}, $Result_{batch}$, ε, γ, K) ;
    partialResults $\longleftarrow$ partialResults $\cap$ keyValueRadixSort($Result_{batch}.scores$, $Result_{batch}.ids$)[:K];
end
return keyValueRadixSort(partialResults.scores, partialResults.ids)[:K];

Algorithm 5: Top-k SDJ hash kernel.

input : $points$, ε
output: $pointGridHash$
index $\longleftarrow$ (blockIdx.x * blockDim.x) + threadIdx.x;
if $index > |points|$ **then**
    return;
gridCell $\longleftarrow$ pointToCell(points[index], ε);
pointGridHash[index] $\longleftarrow$ spaceFillingCurveHash(gridCell);

Algorithm 6: Top-k SDJ filtering kernel.

```
   input  : R, hashValuesR, S_batch, Result_batch, ε, γ, K
   output: Result_batch
 1 index ⟵ (blockIdx.x * blockDim.x) + threadIdx.x ;
 2 heap ⟵ ∅ ;
 3 if index >= |S_batch| then
 4 |   return;
 5 s ⟵ S[index] ;
 6 gridCellS ⟵ pointToCell(s, ε);
 7 foreach x ∈ {−1,0,1} do
 8 |   foreach y ∈ {−1,0,1} do
 9 |   |   foreach n ∈ {−1,0,1} do
10 |   |   |   pointHash ⟵ spaceFillingCurveHash(gridCell + (x, y ... n));
11 |   |   |   limit_lower, limit_upper ⟵ equalRang(hashValuesR, pointHash);
12 |   |   |   foreach r ∈ {R[limit_lower]...R[limit_upper]} do
13 |   |   |   |   if withinDistance(s, r, ε) ∧ γ(r.score, s.score) > heap.minValue then
14 |   |   |   |   |   localHeap.insert((r.id, s.id, γ(r.score, s.score)));
15 |   |   |   end
16 |   |   end
17 |   end
18 end
19 Result_batch ⟵ Result_batch ∩ localHeap ;
```

Algorithm 4 shows the pseudo code of the new top-k SDJ algorithm.

Line 1 to 4 finds the hash values from both R and S. The kernels are launched with one thread for each point in the respective datasets. The time complexity is $\mathcal{O}(|R|+|S|)$ or alternatively $\mathcal{O}(|R|*d+|S|*d)$ if the hash function depends on the dimensionality of the space as it often does. Sorting is done using radix sort on unsigned integers of 32 bits. The algorithm is a parallel algorithm designed for GPUs described in [24]. The time complexity is $\mathcal{O}(|R|+|S|)$ for this step.

Line 5 logically splits S into appropriate batches. The batch size is calculated from the space allocated to the result based on a user parameter. The time complexity is constant $\mathcal{O}(1)$ as we only need to calculate $\left\lceil \frac{|S|}{batchSize} \right\rceil$ and not physically split the data into separate memory locations.

Line 7 and 8 runs the batches through through the top-kFilterKernel. The filter kernel launches one thread for each point in the current batch. Comparing the point in S to all points in the surrounding cells have time complexity $\mathcal{O}(|S|\log(|R|)+|S|\frac{3^d\varepsilon}{V_R}|R|K)$ where V_R is the total volume R covers and $3^d\varepsilon < V_R$.

An important observation is that the time complexity is largely dependent on $\frac{3^d\varepsilon}{V_R}$. If we have epsilon that covers the whole space the complexity is actually worse than quadratic. But since it is reasonable to assume that ε is a small factor the complexity is effectively $\mathcal{O}(|S|\log(|R|)$ for each thread. Since the parallelization factor is high when running on the GPU its maybe reasonable to expect the performance to be close to an algorithm with linear runtime.

Line 9 sorts the resulting top-k values from the batch using radix sort. The time complexity is $\mathcal{O}(BatchSize*K)$ Since the batches are small in size and not memory transfers are needed this step is expected to account for a very small part of the total runtime. This is also true for

the final step in the algorithm at Line 11 that sorts all the top k values in the partialResult array which does also have linear time complexity.

Chapter 6

Methodology

This chapter will describe the methodology used to benchmark and profile the spatial join methods. The synthetic datasets are presented in Section 6.1 and real data is presented in Section 6.2. The environment setup in Section 6.3 describes the computer setup used when benchmarking for reproducible results and easier comparison. Profiling tools for profiling CUDA code is briefly discussed in Section 6.4. Finally implementation of the benchmarker and join methods are discussed in Section 6.5.

6.1 Uniform synthetic data

6.1.1 Box data

The uniform synthetic box data is generated based on density. The generator takes as input the number of boxes, dimension minimum and maximum values to make a closed space and density. It then adjusts the volume of the boxes to get the desired density. Density p is in this case defined as the sum of the volume the objects cover divided by the total volume of the space they are contained in.

$$p = \frac{\sum V_o}{V_s} \tag{6.1}$$

, where V_o is the volume of a spatial object and V_s is the volume of the closed space.

Table 6.1 shows the combinations used when benchmarking.

Table 6.1: The properties of the uniform box datasets.

Distribution	Dimensions	Density(p)	Cardinality	Coordinate range
Uniform	2,3,4,5	0.1, 1, 10	10^2 to 10^7	0 to 1

6.1.2 Point data

Synthetic point data was generated using the same uniform distribution as for boxes. All coordinates are represented by single precision floating point numbers and will range between 0 and 1. Since the definition of density used for box data does not make sence for point data we have removed it from the table. Carnality will be a better indicator of the ”density” of the point dataset. Table 6.2 show the properties of the datasets used for benchmarking point data.

Table 6.2: The properties of the uniform point datasets.

Distribution	Dimensions	Cardinality	Coordinate range
Uniform	2	10^3 to 10^8	0 to 1

6.2 Real data

The real datasets are a collection of mapping data gathered from open data platforms. Table 6.3 describes the properties of the datasets. Figure 6.1 shows a visualization of all the subway entrances in New York together with all roof top water tanks as a representation of how the datasets may look like. The datasets used when benchmarking were aggregated datasets were the geographical positioning data was first transformed into the World Geodetic System (WGS), a standard use in cartography. WGS84 is the revision used.

Table 6.3: Real geographical point datasets used in benchmarking.

Name	Cardinality	Description
Harbour water quality	89545	Map of all administrative states an provinces in the world version 4.1.0 from naturalearthdata.com
Radioactive equipment	6444	Map of all urban areas derived from the MODIS satellite from 2002 to 2003. Version 4.0.0 from naturalearthdata.com
Trees	683788	Map of all census block groups in USA from arcgis.com.
Roof top water tanks	25054	Map of all water bodies in USA from arcgis.com
Points of interest	96	Map of areas in the world subject to floodings in 2002. The map is not complete but around 25%. Available at maps.princeton.edu/catalog/
Subway entrance	1928	Map of all building footprints in New York from 2016. Available at opendata.cityofnewyork.us
Address	964214	Map of all New York tax lots from 2008. Available at opendata.cityofnewyork.us

6.3 Setup environment

The environment used for benchmarking was two Intel Xeon Gold 5120 CPU at 2.20GHz with 14 codes 28 threads each with 125 GB main memory. The operating system used was Linux Ubuntu 18.04.4 LTS. To make the benchmarking probram CMAKE 3.16.4 was used to generate make files. For compilation the system used GCC 7.5.0 and g++ 7.5.0.

The graphics cards used by the system was a Tesla P100 PCIe 3 x16 using CUDA 9.2 compiled for compute architecture 6.0. The specifications of the Tesla P10 card can be found in Figure 6.4

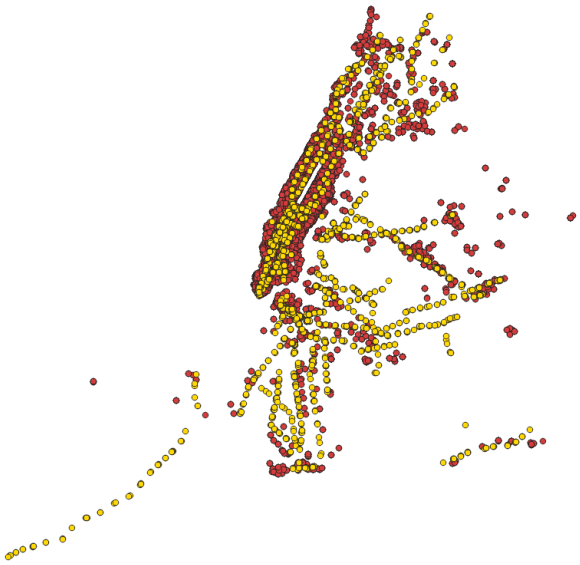


Figure 6.1: Visualization of all subway entrances in New York(yellow) and all roof top water tanks(red).

6.4 Profiling

Profiling code on the GPU cannot be done the same way as on CPU. Typical tools as VALGRIND [25] an PAPI [26] does not work when you want to profile CUDA GPU programs. Therefore NVIDIA Nsight Systems(NNS)[27] is used to gather detailed information about how the algorithms perform. We have used NNS to gather information as occupancy, SU utilization, memory statistics and optimization possibilities. Some memory statistics as overall memory usage is still gathered by the benchmaker we created.

Figure 6.2 shows a histogram of 1000 runtimes for the bruteforce nested loop spatial join algorithm. When benchmarking runtime for the algorithms we take the mean of 20 runtime samples.

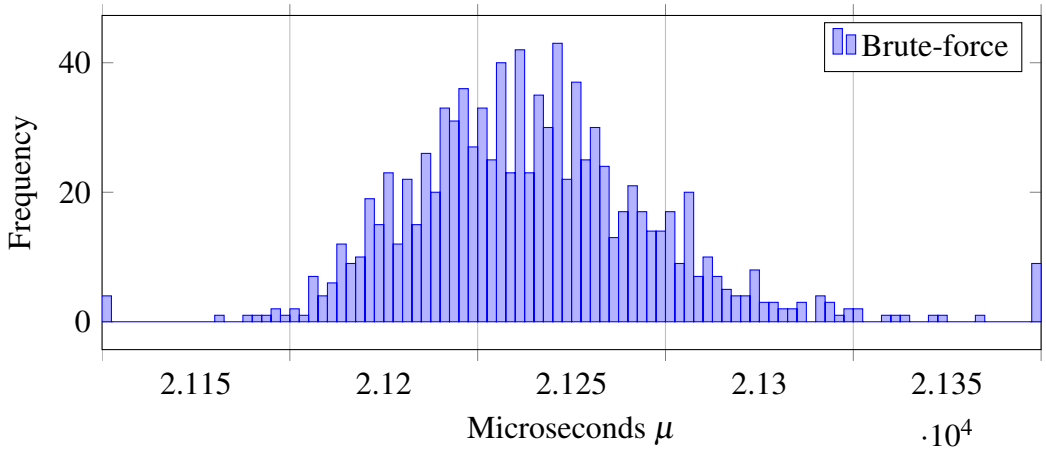


Figure 6.2: The runtime for the problem size 10^5 x 10^4 on a uniform dataset. The figure shows 1000 runtimes recorded.

Table 6.4: Some important specifications of the NVIDIA Tesla P100 PCIe 12GB.

Device property	Value
Name	Tesla P100 PCIe 12GB
Compute capability	6.0
Base clock frequency	1126 MHz
Boost clock frequency	1303 MHz
Memory type	HBM2
Memory bandwidth	549 GB/s
Global memory	12GB
Streaming multiprocessors	56
CUDA cores per SM	64
Cuda codes total	3584
Constant memory	64 KB
Shared memory(per SM)	64 KB
32-bit registers(per SM)	64KB
Warp size	32 threads
Max grid dimensions	x:2147483647 y:65535 z:65535
Max block dimensions	x:1024 y:1024 z:64
Max threads in a block	1024
Max resident blocks per SM	32
Max resident warps per SM	32

6.5 Implementation

To benchmark the join methods we have continued developing the benchmarker Truls Rustad Fossum developed for his master thesis on ”Efficient Spatial Search using Memory Resident R-trees”. The benchmarker is rewritten to be able to handle benchmarking of Spatial join methods for both box and point data. The methods are compiled separately from the benchmarker as shared objects. The benchmarker loads the shared objects and run the join methods. The user need to specify what type of data is being joined when compiling. When running the benchmarker the user provides the R and S datasets and what information they want to record eg. average runtime, all runtimes or memory usage statistics.

The benchmarker is written in C++11 and join methods are written using CUDA 9.2. Both the benchmarer and join method code is compiled using the -03 optimization parameter. For the CUDA code this only affects the code that is running on the CPU not the GPU. The nvcc compiler that compiles the CUDA code uses mostly default parameters except that the paramers ”-use-fast-math” and ”-Xptxas -dlcm = ca” are used to ensure optimized floating point operations and the use of l1 cache for global loads respectively.

The Thrust library [23] was used for binary search and key value radix sorts. Thrust is a CUDA library that provides highly optimized implementations of some common algorithms. All radix sort calls were executed using the ”thrust::device” execution policy so the algorithms were running in parallel on the GPU. The binary sort executions that were executed from a GPU kernel used the ”thrust:seq” execution policy for serial execution.

Heavy use of templates allows the compiler to do optimizations by increased knowledge by doing loop unrolling on dimensional loops or knowing how much memory to allocate for

the result set. The result set size can be unknown before running the join. Therefore there is a compile option to set the result set size. This prevents us from having to dynamically allocate memory for the join method while it is running. If the result set is larger than the allocated memory the program will prompt the user to allocate more space. If batching and result size estimation is used this parameter is ignored. All spatial data is read as 4 byte floating point numbers.

Chapter 7

Analysis and Discussion

In this chapter we will benchmark the new top-k SDJ algorithm and compare it against a simple baseline algorithm which is explained in Section 7.1. Section 7.2 will present the benchmark results before we discuss the suitability for a top-k SDJ algorithm on GPU in Section 7.3. The benchmarking methodology is explained in Chapter 6. The details of the uniform point data is shown in Section 6.1.2 while the physical setup, profiling tools and implementation details can be found in Section 6.3, 6.4 and 6.5 respectively.

7.1 Baseline algorithm

For comparison we made a baseline algorithm. Since there are no other algorithms solving the same problem on GPU. The purpose is to compare the algorithm described in Section 5.1 to a simpler method where we first find the points within ε distance as a filtering step before the refinement step which sorts these points using radix sort on GPU to get the top-k values. The method for calculating the pairs is very similar to the Algorithm 6. The only difference is that no local heaps need to be updated since the sorting happens afterwards. The pseudo code for the baseline algorithm step can be seen in Algorithm 7.

As Algorithm 8 show we need to update an index value for each time we get a new result we that we want to be a part of the final refinement radix sort. Since this is a shared global array between all threads we need to use an atomic add to increment the result index position so no

results overwrite each other.

Algorithm 7: The baseline top-k SDJ algorithm

input : R, S, ε, γ, K, *ResultSize*
output: *Result*

```
1  hashValuesR ⟵ hashKernel(R, ε);
2  hashValuesS ⟵ hashKernel(S, ε);
3  keyValueRadixSort(hashValuesR, R);
4  keyValueRadixSort(hashValuesS, S);
5  baselineSDJFilter(R, hashValuesR, S, Result, ε, γ, K) ;
6  Result ⟵ keyValueRadixSort(Result.scores, Result.ids)[:K];
7  return Result;
```

Algorithm 8: SDJ filter for used in the baseline algorithm.

input : R, *hashValuesR*, S, *Result*, ε, γ, K
output: *Result*

```
1  index ⟵ (blockIdx.x * blockDim.x) + threadIdx.x ;
2  if index >= |S| then
3  |  return;
4  s ⟵ S[index] ;
5  gridCellS ⟵ pointToCell(s, ε);
6  foreach x ∈ {−1,0,1} do
7  |  foreach y ∈ {−1,0,1} do
8  |  |  foreach n ∈ {−1,0,1} do
9  |  |  |  pointHash ⟵ spaceFillingCurveHash(gridCell + (x, y ... n));
10 |  |  |  limit_lower, limit_upper ⟵ equalRang(hashValuesR, pointHash);
11 |  |  |  foreach r ∈ {R[limit_lower]...R[limit_upper]} do
12 |  |  |  |  if withinDistance(s, r, ε) ∧ γ(r.score, s.score) > heap.minValue then
13 |  |  |  |  |  resultIndex ⟵ atomicAdd(resultIndex, 1) ;
14 |  |  |  |  |  Result[resultIndex] ⟵ (γ(r.score, s.score), (r.id, r.id) );
15 |  |  |  end
16 |  |  end
17 |  end
18 end
```

7.2 Comparison

Figure 7.1 shows the overall runtime of the algorithm when using shared memory and registers. We set ε to 0.001 and K to 16 as default. The block size was 64 threads per block. As the figure shows there is no significant difference between the two methods for a fixed K size. Both algorithms perform well with no sizes exceeding 6 seconds of runtime even for problems where $|R| = |S| = 10^8$. It is worth noting that for large sizes or R the runtime is close to constant until $|S|$ increases to 10^7. Figure 7.2 shows the runtime when $|R| = |S|$. Since the two methods performs so similarly we will only compare the baseline algorithm to the register method. If there are some interesting differences between the register and shared memory methods they will of course be noted.

The runtime will be affected mainly by the ε and K parameters. When we increase epsilon

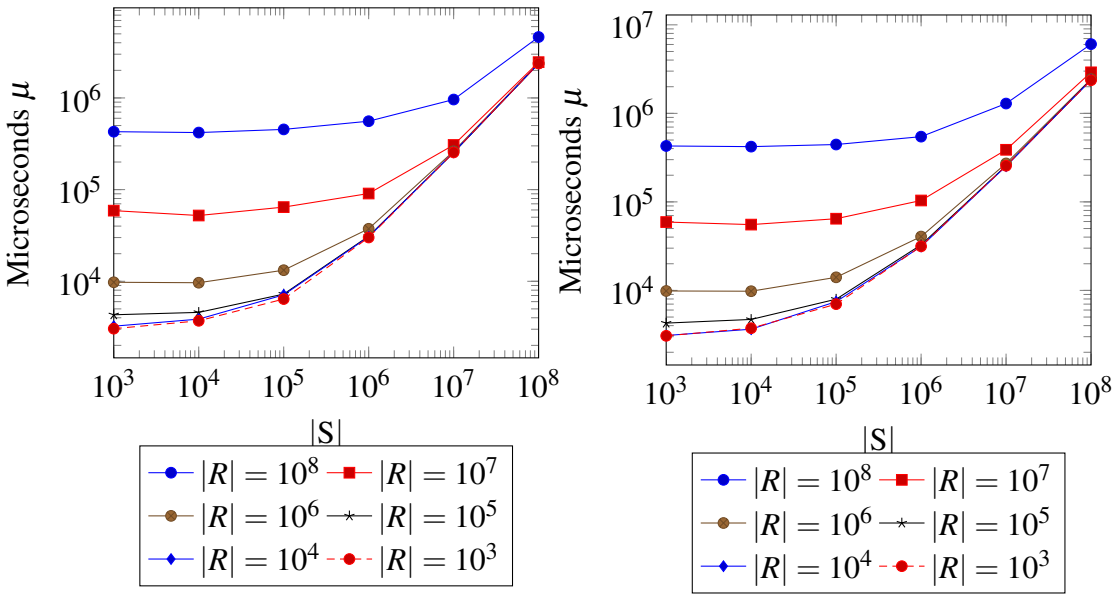


Figure 7.1: (Left) The runtime of top-k SDJ with register memory for the top-k heap at different problem sizes of R. (Right) The runtime of top-k SDJ with register memory for the top-k heap for the same problem sizes. $|R| = |S|$

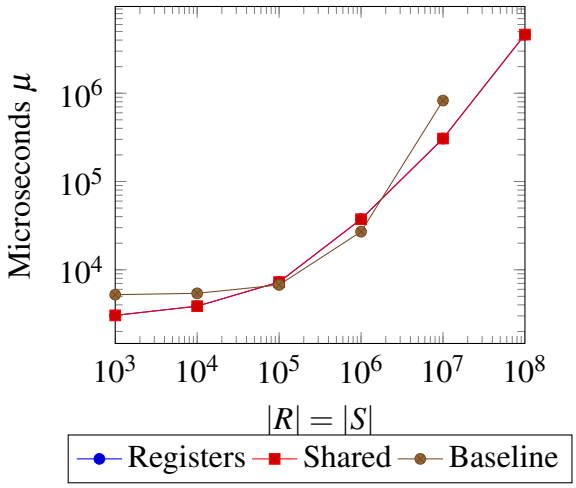


Figure 7.2: The runtime of the top-k SDJ algorithm for problems where $|R| = |S|$. the baseline method could not solve problems larger than 10^7.

the number of points that needs to be evaluated increases drastically. Figure 7.3 shows how the runtime of the register method is affected by ε for some problem sizes. The R set is fixed to size 10^6. Increasing ε shows that the algorithm handles the increase in epsilon exceptionally. This is mainly because the algorithm limits the amount of points sent the the radix sort steps to K. One positive aspect with this is that the algorithm does not have space problems when increasing ε. All sizes of ε are handled nicely.

This is a huge benefit over the baseline method. Figure 7.4 shows how the baseline method compares to the register method when ε increases. The figure show the percentage of time used to filter the points and sorting relative to the total runtime. Note that the percentages does not add to 100% at some time is also used to hash and sort the points using the Hilbert space

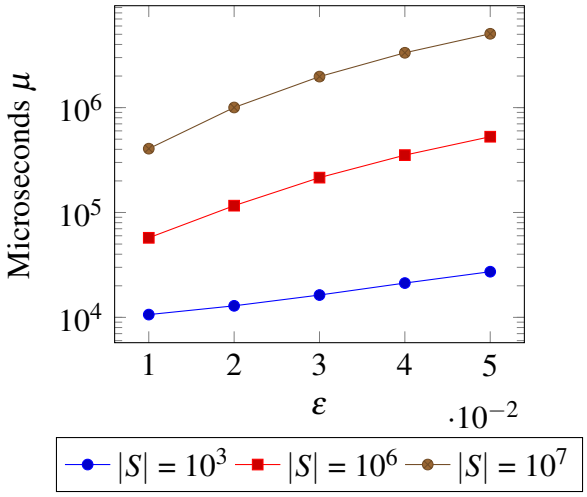


Figure 7.3: The runtime of the top-k SDJ algorithm for different values of ε.

filling curve. We do not show this step since its the same for all methods and during testing it accounted for about 1 % or less of the total runtime. The achieved occupancy is also shown as marks in Figure 7.4.

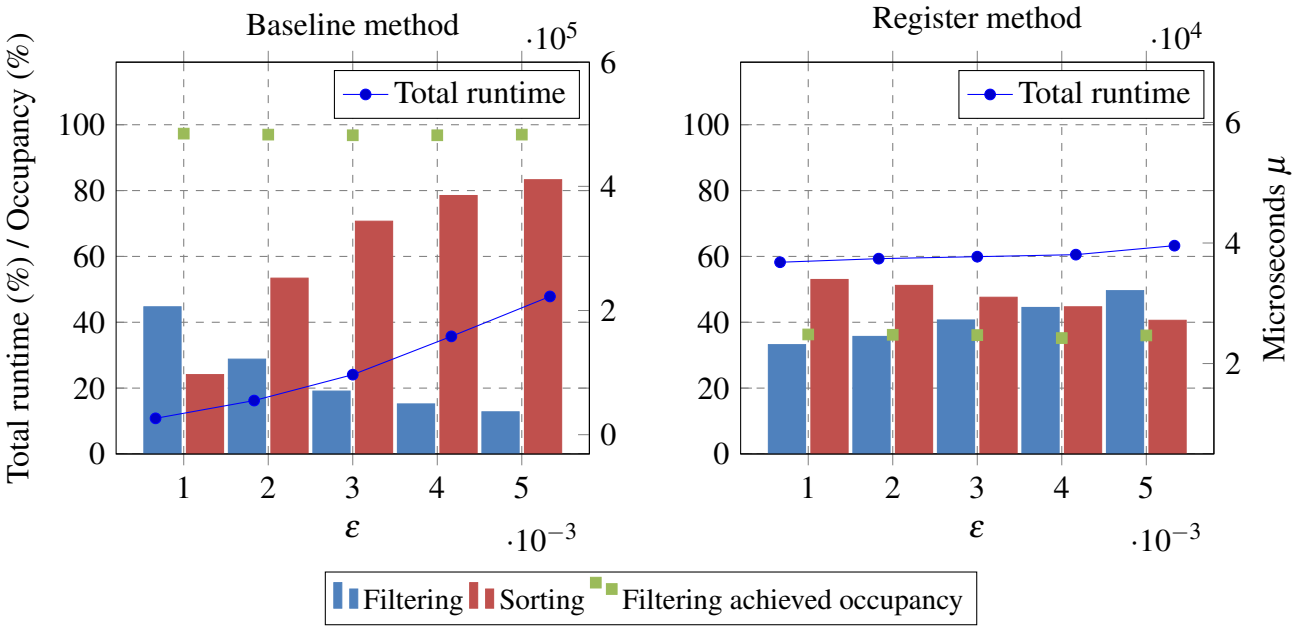


Figure 7.4: (Left) The percentage of the total runtime the baseline method used to filter or sort when ε is increasing. (Right) The same benchmark as on the left but with the top-k SDJ register method instead.

The baseline gets in trouble if too many pairs are found in the filtering step before sorting. In our test we got 313893485 results from this step for problem sizes of 10^7 x 10^7 points with ε = 0.001. This was equivalent to about 3.7 GB of data that needed to be sorted. The impact of this is clearly visible in Figure 7.4. The time used to sort the points after filtering dominates the total time used when ε increase. This is in sharp contrast to the register method where we can see that the filtering step is the one increasing. This is natural since the number of point pairs sent to sorting in the register method is constant. When ε increase the filtering step will therefore naturally take more time relative to the sorting step. The runtime evolves much better for the

register method and shows that the ε parameter is handled much better than in the baseline method.

Another interesting observation is the achieved occupancy. Even tough the filtering step in the baseline method has much better achieved occupancy than the register method the runtime is much worse. This tells us that we actually benefit from sacrificing some occupancy in in trade of a lighter sorting step.

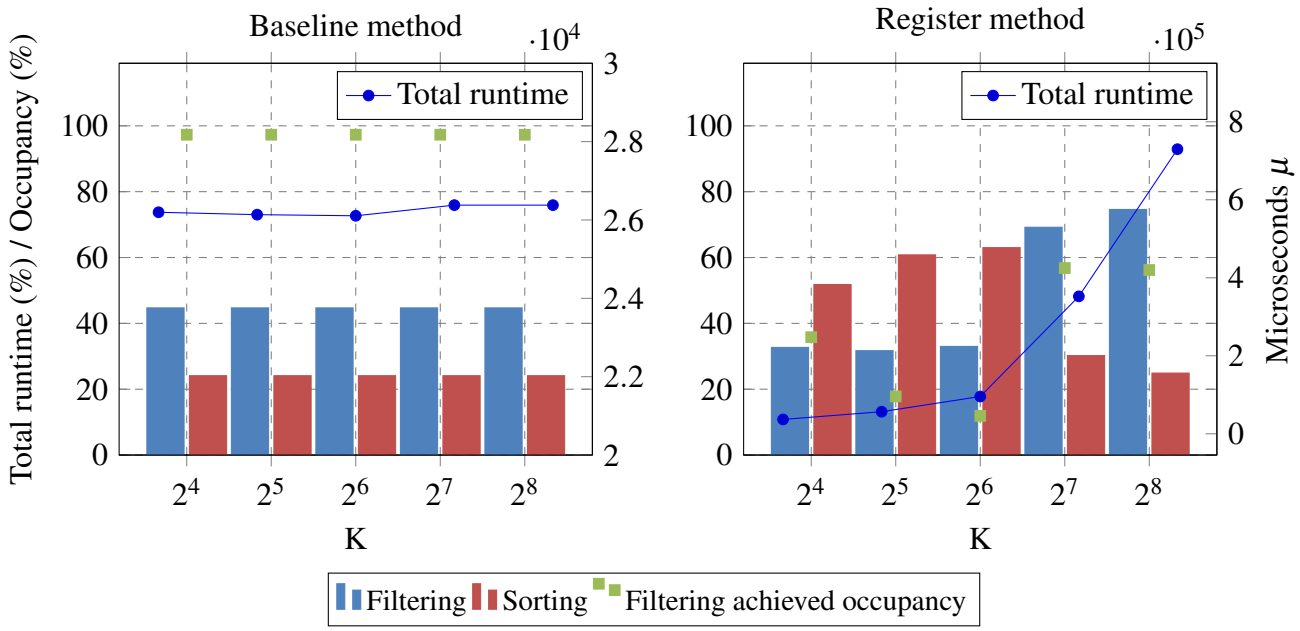


Figure 7.5: (Left) The percentage of the total runtime the baseline method used to filter or sort when K is increasing. (Right) The same benchmark as on the left but with the top-k SDJ register method instead.

Increasing the K parameter tells another story. Since the baseline method returns all points within ε distance to be sorted the runtime is not affected by K as shown in Figure 7.5. The time used on filtering and sorting is relatively balanced.

The register method is by contrast much more effected. When K increase the amount of registers used in each kernel also increase. This negatively effects the achieved occupancy on each SM since all registers in a block needs to be allocated. There will therefore be assigned very few blocks at a time which is problematic for memory latency hiding. This is true until K = 64. After that the compiler decides to use local memory instead which increase the runtime significantly. This is explained in detail in Section 2.3.2. Figure 7.5 show that after K = 64 we get a dramatic increase in total runtime. This is caused by the increased time used by the filtering step. The runtime evolves much better using the baseline method and shows that the ε parameter is a significant drawback in the register method.

The achieved occupancy is as expected high for the baseline method. The register method do by contrast struggle to achieve respectable occupancy. When we increase K up to 64 only registers are used for the local heaps. The number of blocks the SMs can be assigned are is therefore extremely low. For K = 64 we could only allocate 4 blocks per SM. When K is larger than 64 local memory is also used to store the local heaps. The occupancy therefore increases as the SMs can allocate memory for more blocks. In our testing the register usage was still the limiting factor. Each SMs could only locate 20 out of a maximum of 32 blocks per SM for K larger than 64 where local memory was used.

Figure 7.6 shows also shows how the algorithms are affected by K but not the shared memory method is also included. The shared memory is not large enough to handle more than 64KB so the algorithm as it is right now cannot run problems with K $>$ 64 when using shared memory. Using registers it seems that it is possible to solve larger problems, but that is with using local memory instead of registers. Since local memory is much slower we see a significant increase in runtime.

Using registers is therefor better than shared memory since it can handle larger problem sizes, but at the cost of significant runtime increase for large values of K.

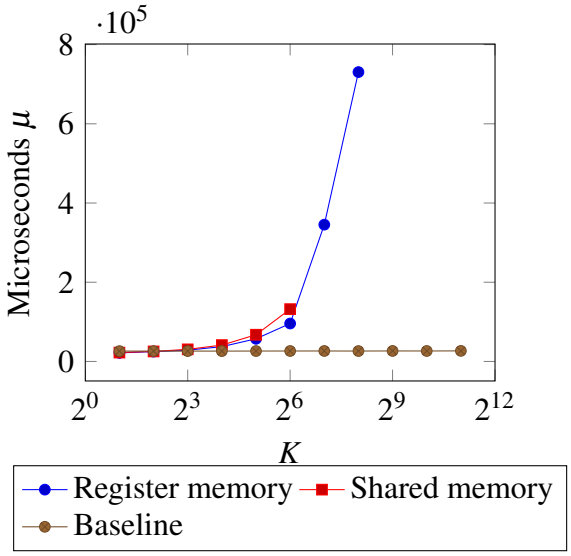


Figure 7.6: The runtime of top-k SDJ for different sizes of K with ε = 1 and the problem size $|R| = |S| = 10^6$.

The memory effect of increasing K can be seen in more detail in Figure 7.7. The figure shows how memory reads and writes changes when we increase the K parameter from K=32 to K=64 where local memory is used. Arrows pointing to the right are writes and the arrows pointing left are reads. All green boxes are logical memory spaces while blue boxes are physical memory chips. The ”Unified Cache” box is the same as he L1 cache layer discussed in Section 2.3.1. The figure only shows memory usage for the algorithm using registers as the difference between it and the shared memory method is that the shared memory usage would be visible.

The cache hit rate is shown as a percentage in the ”Unified cache” and ”L2 cache” boxes. They show that the hit rate is relatively high for both problems. The diagram showing cache usage for K=32 have 78% hit rate in L1 cache compared to 68% for the K=64 problem. Despite the L1 cache being higher the L2 cache hit rate is higher for the K=64 problem. This might be because the heaps that gets stored in local memory now also utilizes the L1 cache taking up space from the global memory requests in L1 memory.

Figure 7.8 shows how the register algorithm behaves running the join on a collection of real datasets.The most interesting thing to note is that the harbor water quality datasets joined with the tree dataset have an interesting jump in runtime after epsilon exceeds 10^{-2}. This is because for lower values of epsilon there are very few data points that intersect since the two datasets are very spatially separated. No trees are planted in the water. Therefor when epsilon increases so that the radius around the trees reaches out to the water the runtime also increases significantly. The other datasets does not have the same problem as the spatial range or the

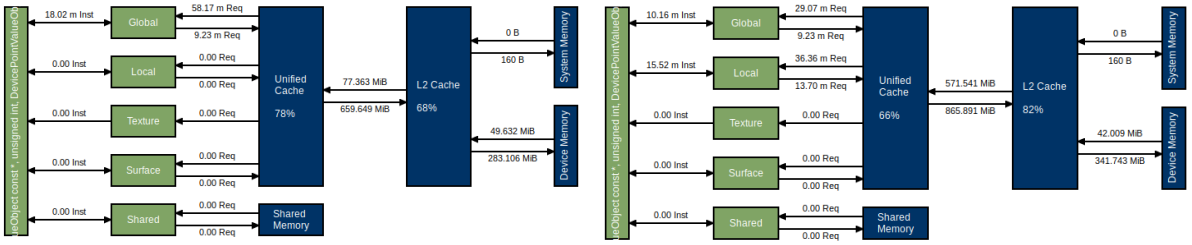


Figure 7.7: (Left) The top-k SDJ algorithms memory usage patterns for K=32, $\varepsilon = 1$. (Right) The same as the left figure but now K=64. The local memory is now being used. Both figures are generated in the NVIDIA Nsight profiler.

objects are overlapping much more. The last Figure 7.9 confirms that the memory usage is balanced between R and S.

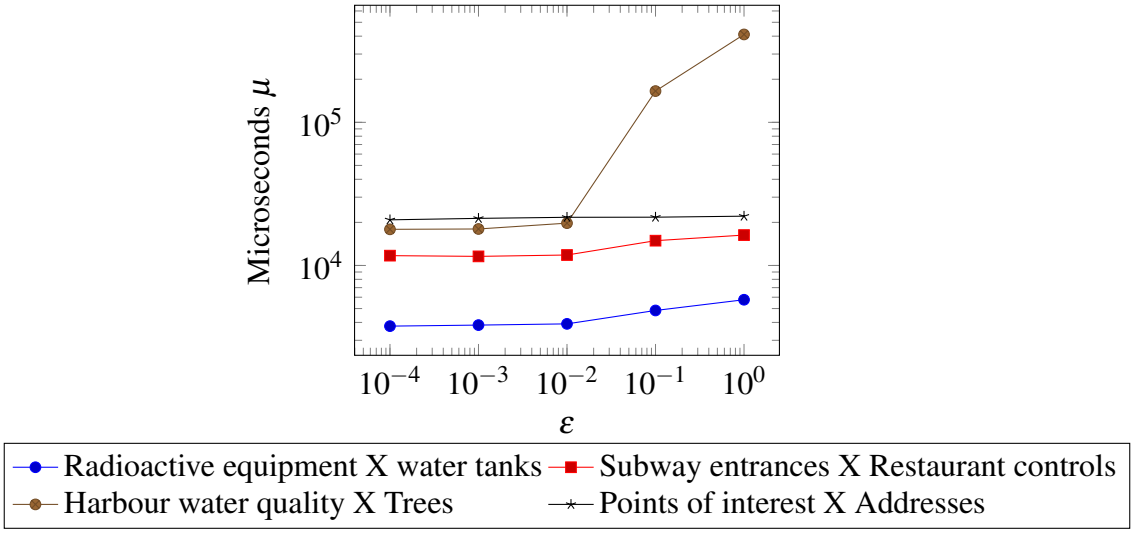


Figure 7.8: The runtime of top-k SDJ for low ε values for the problem size $|R| = 10^6$.

7.3 Suitability for GPU architecture

There are many factors that contribute to the evaluation of how suitable the GPU architecture is for solving the top-k SDJ problem. The main problem we faced was finding a good way to handle the top-k part.

The new top-k SDJ algorithms we designed solved this problem well for $K < 64$. The fact that we could use K to reduce the number of pairs that needed to be sorted both reduced the space complexity and memory transfer times greatly which resulted in a very efficient solution. The main problem is that when K increases the limited register and shared memory available makes the GPU use slow global memory instead. Our analysis show that a baseline method that first filters all points and then sorts them for the top-k will perform much better for large K values. For this approach to be better for GPU we would need more register or shared memory.

The spatial distance join part of the problem is much easier to solve efficiently on the GPU. By using a grid index we can reduce the serach space drastically while still being able to utilize the GPU hardware to its full potential. This is in large because the nature of the problem is

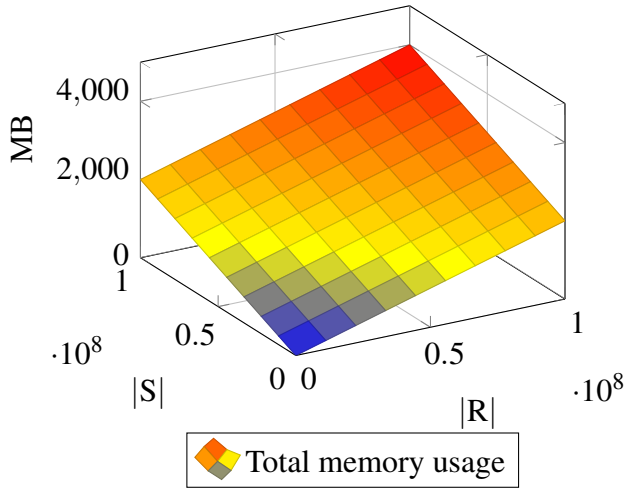


Figure 7.9: The memory usage of top-k SDJ when for all problem sizes. the space used increace linear ly with the input size. This is with 4 byte floats for points in 2D with 4Bytes unsigend integers for scores and batch size of 50 MB.

more parallelization friendly. All calculations that find points within ε distance are independent of each other. No need for common structures or communication between the threads.

This is the main problem we face when we add the top-k part to the problem. Since the calculations are not independent we need common structure such as a heap if we want to solve the problem in one step. Solving the problem in two steps as the baseline method does lets the problem be solved utilizing the GPU hardware to its full potential. The drawback is that the amount of point pairs that needs to be sorted includes a huge amount of unnecessary points. This also makes the space complexity a much greater issue than in our top-k SDJ methods using registers.

The conclusion is therefor that the top-k part of the problem is difficult to take advantage of on GPU while still utilizing the massively parallel hardware without significant drawbacks.

Chapter 8

Conclusion and future work

Recently advancements in GPU and massively parallel architectures have lead to an increased interest in many fields. Even though there is relatively limited research done utilizing the GPU for spatial join operations we think this architecture will get an increased importance as more research is done.

In this thesis we explored how the GPU architecture could best be utilized for indexing spatial data. We explored existing methods and identified challenges that occur because of heterogeneous computing scheme and memory layout of the GPU. The methods were tackling different spatial problems to show how other researchers have tackled similar problems. We developed a new top-k SDJ algorithm based on the knowledge we gathered from studying existing indexing and spatial join methods on GPU.

Finally we conducted experiments to study the new top-k SDJ algorithm to get a better understanding of the challenges and its performance.

8.1 Conclusion

The thesis started out by asking one main research questions and two sub reserach guestions. Based on the knowledge gathered and observations we have done this section will discuss if the research questions are answered.

The main research question **RQ1** asked if GPUs are suited for top-k spatial distance join queries. To be able to answer this we needed to answer two sub research questions.

SRQ1 : Which spatial indexes and data types if any are suited for GPU?

In Chapter 3 we went through some common spatial indexing techniques and identified the some challenges with traditional hierarchical indexes as R-trees that are common for CPU. The common consensus in the research papers we have read and our own conclusion from that chapter is that a uniform grid structure or ε grid is a good choice for our top-k SDJ algorithm. Uniform grids can also be used for other spatial join problems as our investigation of spatial join in Chapter 4 show that the grid structure can also be used on box data as well as point data for other queries as self join. Chapter 4 also gave us knowledge that not materializing the grid structure could be a good idea for our top-k SDJ algorithm.

SRQ2 : How can top-k be efficiently implemented in the GPU memory architecture?

SRQ2 is answered extensively while designing the top-k SDJ algorithm in Chapter 5. The conclusion we come to in this thesis is that we have two options for top-k on GPU. One use shared memory or registers to hold local top-k heaps while the other use a filter then sort approach. Both methods are limited by the memory size available and the low occupancy when using large amounts of register or shared memory on the GPU. In Chapter 7 we show that both methods perform similarly until K = 64. After that the shared memory method does not work as there are no space left. For the register method local memory is used for large Ks which effects the performance negatively. The baseline filter then sort method performed the best when we increase K but run into space issues when there are large amounts of pairs returned from the filtering step. So the the answer to **SRQ2** is that top-k can be efficiently implemented by local top-k heaps for $K < 64$ while for larger K values we were not able to find a good approach.

We can now answer **RQ1**. The answer of **SRQ1** tells us that there are spatial structures that are both well suited for the SDJ part of the problem and suited for the GPU architecture. The answers we got from **SRQ2** on the other hand shows that we could not find a method without significant drawbacks for large K values. We will therefor conclude that GPU is well suited for SDJ but we need more research to find better ways of handling the top-k part of the problem.

8.2 Future work

There is still a lot of research that could be done on spatial join problems running on GPU. The problem of hierarchical index structures is being researched already but there seems to be some uncertainty of how well its suited for GPU. A comparison of how different indexing structures perform on the GPU could be interesting as our conclusions may not necessarily be the right ones as we were not able to implement and compare them.

It could also be interesting to look at how ε grids could be used in a spatial distance query where the data was boxes instead of points. The approach used by the MLG-join algorithm in Section 4.2 shows that grids are suited for intersection joins. Maybe a similar approach could be used or distance joins?

Doing different join predicates could also be interesting. Our top-k SDJ algorithm uses distance less than ε as to compare the objects from R and S. It could be interesting to see how for example intersection between boxes or polygons could could be implemented as a top-k spatial intersection join algorithm.

A better solution to handle large K values for the top-k SDJ algorithm would also be a research topic worth investigating further. Our approach utilized the on chip memory banks to utilize the high speed reads and writes. A method that find a more space efficient approach for local heaps or do not use local heaps at all could be of interest.

Bibliography

[1] S. Qi, P. Bouros, and N. Mamoulis. Efficient top-k spatial distance joins. In *International Symposium on Spatial and Temporal Databases*, pages 1–18, 2013.

[2] S. Qi, P. Bouros, and N. Mamoulis. Efficient top-k joins on complex data types. 2015.

[3] Naga Govindaraju, K, Brandon Lloyd, Wei Wang, Ming Lin, and Dinesh Manocha. Fast computation of database operations using graphics processors. In *Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, pages 215–226, 2004.

[4] Antonin Guttman. R-trees: A dynamic index structure for spatial searching. In *SIGMOD'84, Proceedings of Annual Meeting, Boston, Massachusetts, USA, June 18-21, 1984*, pages 47–57, 1984.

[5] Jianting Zhang, Simin You, and Le Gruenwald. Parallel spatial query processing on gpus using r-trees. In *Proceedings of the 2nd ACM SIGSPATIAL International Workshop on Analytics for Big Geospatial Data, BigSpatial@SIGSPATIAL 2013, Nov 4th, 2013, Orlando, FL, USA*, pages 23–31. ACM, 2013.

[6] Sushil K. Prasad, Michael McDermott, Xi He, and Satish Puri. Gpu-based parallel r-tree construction and querying. In *2015 IEEE International Parallel and Distributed Processing Symposium Workshop, IPDPS 2015, Hyderabad, India, May 25-29, 2015*, pages 618–627. IEEE Computer Society, 2015.

[7] Jinwoong Kim and Beomseok Nam. Co-processing heterogeneous parallel index for multi-dimensional datasets. *Journal of Parallel and Distributed Computing*, 113:195–203, 2018.

[8] Phillip G Ward, Zhen He, Rui Zhang, and Jianzhong Qi. Real-time continuous intersection joins over large sets of moving objects using graphic processing units. *The VLDB Journal—The International Journal on Very Large Data Bases*, 23(6):965–985, 2014.

[9] Michael Gowanlock and Ben Karsin. Gpu accelerated self-join for the distance similarity metric. In *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 477–486. IEEE, 2018.

[10] Danial Aghajarian, Satish Puri, and Sushil K. Prasad. GCMF: an efficient end-to-end spatial join system over large polygonal datasets on GPGPU platform. In *Proceedings of the 24th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems, GIS 2016, Burlingame, California, USA, October 31 - November 3, 2016*, pages 18:1–18:10, 2016.

[11] Jianting Zhang, Simin You, and Le Gruenwald. Parallel selectivity estimation for optimizing multidimensional spatial join processing on gpus. In *ICDE*, pages 1591–1598. IEEE Computer Society, 2017.

[12] Vebjorn Ljosa and Ambuj K. Singh. Top-k spatial joins of probabilistic objects. In *Proceedings of the 24th International Conference on Data Engineering, ICDE 2008, April 7-12, 2008, Cancún, Mexico*, pages 566–575. IEEE Computer Society, 2008.

[13] Hamza Mustafa, Eleazar Leal, and Le Gruenwald. Fasttopk: A fast top-k trajectory similarity query processing algorithm for gpus. In *2018 IEEE International Conference on Big Data (Big Data)*, pages 542–547. IEEE, 2018.

[14] Anil Shanbhag, Holger Pirk, and Samuel Madden. Efficient top-k query processing on massively parallel hardware. In *Proceedings of the 2018 International Conference on Management of Data*, pages 1557–1570, 2018.

[15] P. Bouros and N. Mamoulis. Spatial joins: what's next? *SIGSPATIAL Special*, 11:13–21, 2019.

[16] George Roumelis, Antonio Corral, Michael Vassilakopoulos, and Yannis Manolopoulos. New plane-sweep algorithms for distance-based join queries in spatial databases. *GeoInformatica*, 20(4):571–628, 2016.

[17] John Von Neumann. First draft of a report on the edvac. *IEEE Annals of the History of Computing*, 15(4):27–75, 1993.

[18] Simon Green. Particle simulation using cuda. *NVIDIA whitepaper*, 6:121–128, 2010.

[19] Yi He, Andrew E Bayly, Ali Hassanpour, Frans Muller, Ke Wu, and Dongmin Yang. A gpu-based coupled sph-dem method for particle-fluid flow with free surfaces. *Powder technology*, 338:548–562, 2018.

[20] Panagiotis Bouros, Shen Ge, and Nikos Mamoulis. Spatio-textual similarity joins. *Proceedings of the VLDB Endowment*, 6(1):1–12, 2012.

[21] David Hilbert. Über die stetige abbildung einer linie auf ein flächenstück. In *Dritter Band: Analysis· Grundlagen der Mathematik· Physik Verschiedenes*, pages 1–2. Springer, 1935.

[22] Michael Gowanlock, Cody M Rude, David M Blair, Justin D Li, and Victor Pankratius. Clustering throughput optimization on the gpu. In *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 832–841. IEEE, 2017.

[23] Nathan Bell and Jared Hoberock. Thrust: A productivity-oriented library for cuda. In *GPU computing gems Jade edition*, pages 359–371. Elsevier, 2012.

[24] Nadathur Satish, Mark Harris, and Michael Garland. Designing efficient sorting algorithms for manycore gpus. In *2009 IEEE International Symposium on Parallel & Distributed Processing*, pages 1–10. IEEE, 2009.

[25] Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. *ACM Sigplan notices*, 42(6):89–100, 2007.

[26] Dan Terpstra, Heike Jagode, Haihang You, and Jack Dongarra. Collecting performance data with papi-c. In *Tools for High Performance Computing 2009*, pages 157–173. Springer, 2010.

[27] NVIDIA. Nvidia Nsight Systems. `https://developer.nvidia.com/nsight-systems`, 2020. [Online; accessed 30-May-2020].

NTNU

Kunnskap for en bedre verden