Master's thesis

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

Erlend Åmdal

Top-k Spatial Join on GPU

Master's thesis in Computer Science
Supervisor: Kjetil Nørvåg
June 2020

NTNU
Norwegian University of
Science and Technology

Erlend Åmdal

Top-k Spatial Join on GPU

Master's thesis in Computer Science
Supervisor: Kjetil Nørvåg
June 2020

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

Problem description

In this project, the aim is to study top-k spatial join and how to execute this query efficiently on GPUs.

Supervisor: Kjetil Nørvåg

Abstract

Given two sets of spatial objects where each object is assigned a score, a spatial join predicate (such as a point distance threshold), and an aggregate function that combines scores for pairs of objects (such as a weighted sum), the top-k spatial join query joins the sets by the spatial join predicate and returns the k pairs with the best scores. Spatial data sets can be complex and large, requiring the use of spatial indexing data structures such as the R-tree to accelerate spatial queries and spatial joins. A number of top-k spatial join algorithms based on R-trees exist, but they are sequential algorithms whose performance is limited by the use of a single thread. To accelerate top-k spatial joins further, parallel algorithms could be applied, but research on this particular subject is limited.

Graphics processors are becoming commodity hardware, and their massively parallel processing capabilities can be used to achieve high performance. APIs such as CUDA have enabled the trend of general-purpose computing on graphics processors (GPGPU), which has become a popular way to accelerate applications in certain domains. Special application design is required to efficiently utilize graphics hardware, which requires new research to adapt to this new computing paradigm. Research shows that parallel spatial queries and spatial joins on R-trees can be sped up significantly with GPGPU, but little research has been made into processing top-k spatial joins and top-k queries in general with GPGPU.

In this thesis, the primary goal has been to determine how to perform top-k spatial joins using GPGPU and determine if or how it can be used to achieve speedups. A secondary goal applies similarly to parallel top-k spatial joins using multi-threading. To achieve this, we researched GPGPU, the R-tree and related algorithms, ranked joins, sorting algorithms and heaps and developed a single-threaded, a multi-threaded and a CUDA implementation of the Block-based Algorithm, and finally performed an experimental evaluation. The experimental evaluation shows that top-k spatial joins can be performed efficiently using GPGPU, but the implementation only achieves significant speedups for particularly large inputs and small outputs. Multi-threaded top-k spatial joins is a more viable alternative in general, where the multi-threaded implementation outperforms the single-threaded implementation in all experiments except for the ones with the smallest inputs.

Sammendrag

Gitt to sett med romlige objekter der hvert objekt har en rangering, et romlig sammenslåingspredikat (f.eks. en punktdistanse-begrensning), og en summeringsfunksjon som sammenslår rangeringen til et par av objekter (f.eks. en vektet sum), slår top-k spatial join-operasjonen sammen settene med det romlige sammenslåingspredikatet og returnerer de k parene som har best sammenslått rangering. Romlige datasett kan være komplekse og store, som gjør det nødvendig å benytte datastrukturer for romlig indeksering slik som R-treet for å øke ytelsen på spatial join og romlige spørringer. Det finnes algoritmer for top-k spatial join basert på R-trær, men de er sekvensielle algoritmer med begrenset ytelse grunnet bruk av kun én tråd. For å øke ytelsen på top-k spatial join videre kan parallelle algoritmer anvendes, men forskning på dette området er begrenset.

Grafikkprosessorer er i ferd med å bli prisgunstig og utbredt maskinvare, og deres evne for massivt parallell prosessering kan benyttes for å oppnå høy ytelse. APIer som CUDA har muliggjort trenden "general-purpose computing on graphics processors" (GPGPU), som har blitt en populær måte å øke ytelsen til applikasjoner i visse felt. Det kreves spesiell design for å effektivt anvende grafikkprosessorer, derfor kreves det ny forskning for å tilpassse metoder til det nye databehandlingsparadigmet. Forskning viser at ytelsen for parallelle romlige spørringer på R-trær kan økes med GPGPU, men det finnes lite forskning på prosessering av spatial joins og top-k-spørringer generelt med GPGPU.

I denne oppgaven har hovedmålet vært å fastslå hvordan GPGPU kan anvendes for top-k spatial join-spørringer og hvorvidt/hvordan dette kan oppnå økt ytelse. Et sekundærmål angår det samme men ved bruk av flere tråder. For å fastsette dette har vi undersøkt GPGPU, R-treet og relaterte algoritmer, ranked joins, sorteringsalgoritmer og heaps og utviklet en enkelttråds-versjon, en flertråds-versjon og en CUDA-versjon av den Blokkbaserte Algoritmen, og konkludert med en eksperimentell evaluering. Evalueringen viser at GPGPU kan effektivt anvendes for top-k spatial joins, men vår implementasjon oppnår kun betydelig økt ytelse for særlig store inndata med få returverdier. Generelt er bruk av flere tråder et mer effektivt alternativ, da flertråds-versjonen har bedre ytelse enn enkelttråds-versjonen i alle eksperimenter utenom de med de minste inndataene.

Acknowledgements

I wish to thank my supervisor Kjetil Nørvåg for this project proposition and for invaluable assistance in the writing of this thesis and the preceding project.

Contents

List of Figures

List of Algorithms

Chapter 1

Introduction

In many applications with spatial objects, objects have both spatial attributes and non-spatial attributes. As an example, an application such as Google Maps provides points of interest and businesses, primarily with their geographic location, but also with reviews, user ratings and various other metadata. Spatial objects are also recorded in scientific fields such as atmospheric, oceanographic and environmental sciences with measurements of several attributes such as temperature, pressure and seismic activity.

A spatial join retrieves pairs of spatial objects from two different data sets satisfying a spatial predicate, such as a spatial query to retrieve all intersecting pairs of objects or all pairs of objects within a certain distance from each other. As an example, for an overnight visit in Oslo with dinner, we are given a set of all restaurants and a set of all hotels in Oslo, and would like to find pairs of highly rated restaurants and hotels that are within 1000 meters from each other. The amount of restaurants and hotels in close proximity to each other will result in too many pairs to reasonably assess, so a reasonable solution is to limit our search to find only the most highly rated pairs. We can express this as a top-k query, where we would like to find the 10 pairs of restaurants and hotels with the best sum of ratings. Highly rated locations can easily be found by sorting, as can highly rated pairs, but the most highly rated locations are not necessarily found in close proximity to each other. We can also find many pairs of locations that are close to each other using spatial indexing methods, but we may not find the pairs with the best ratings quickly. An efficient solution must be able to use both their locations and their ratings to answer the query.

A spatial join can be an expensive operation because spatial datasets can be complex and large. To increase the performance of spatial queries, spatial indexing data structures such as R-trees are therefore used, which can be used to perform efficient spatial joins. R-tree spatial joins can even be augmented with general methods from ranked joins to efficiently answer top-k queries with only partially evaluated spatial joins, which reduces the amount of processing required to compute the answer.

With hardware trends such as reduced I/O costs, increasing memory sizes and multicore processors and graphics processors becoming commodity hardware, innovation is required to properly utilize the hardware that has become available. One way to increase performance is with parallel algorithms, which is enabled by multicore processors, distributed computing and graphics processors. However, existing sequential algorithms often require careful redesign to exploit parallelism, or entirely new parallel algorithms must be created. A number of new design considerations appear when adapting methods for multiple threads, and especially for graphics processors.

The trend of General Purpose computing on Graphics processors (GPGPU) is enabled by programmable graphics processors using APIs such as CUDA and OpenCL. GPGPU presents an opportunity to achieve massively parallel computation, but not without its limitations. Not all applications can truly be adapted for GPGPU. The architecture of GPUs and CPUs are dissimilar in ways that requires applications to be specially designed to be able to fully utilize the resources of a GPU. The use of GPGPU in certain database operators has been studied and have in some cases been shown to have speedups compared to CPU for operations such as relational joins [9] and spatial joins [20].

Top-k queries, top-k joins and spatial joins have been extensively studied, but joins considering both spatial and score attributes at the same time have received limited attention [17]. There are proven methods for processing top-k spatial joins, but it is believed that they can be made more efficient. One direction for research is evaluating the existing methods for parallel processing using multi-core processors or GPGPU.

In this thesis, the primary goal is to determine how to perform top-k spatial joins using GPGPU and determine if or how it can be used to achieve speedups. A secondary goal applies similarly to parallel top-k spatial joins using multi-threading. This thesis is a continuation of a project that researched and laid the groundwork for a CUDA implementation of the Block-based Algorithm for top-k spatial joins [1]. To continue the work, we have developed a limited but working CUDA implementation of the Block-based Algorithm. Going beyond the scope of the previous project, further research has been done to develop a multi-threaded implementation and a single-threaded implementation of the Block-based Algorithm using many of the same concepts and methods. This allows us to compare both the CUDA implementation and the multi-threaded implementation with their alternatives in an experimental evaluation.

The second chapter covers General-purpose computing on graphics processors (GPGPU). This includes important aspects of GPU architecture that affect the design of GPGPU applications as well as the CUDA programming model. The third chapter covers spatial indexing concepts and methods, which relates to queries about the location, size and shape of objects in space. The R-tree is the main data structure used in this thesis, thus a number of techniques for performing spatial queries and joins using R-trees are described. The fourth chapter covers ranked query concepts and methods. Methods for top-k queries, top-k joins as well as top-k spatial queries are described. The fifth chapter describes algorithms and data structures for sorting data that are useful for top-k queries, and highlights the different methods used on CPU and GPU. Some potential optimizations are also identified in this chapter. The sixth chapter describes single-threaded, multi-threaded and CUDA implementations of the Block-based Algorithm for top-k spatial joins. The seventh chapter contains an experimental evaluation and comparison of all the implementations. Finally, a conclusion and further work is outlined in the final chapter.

1.1 Related Work

Brinkhoff et al. [5] described a method to to perform spatial intersection joins by performing a concurrent descent down two R-trees. The basic principle is that if two entries of different R-trees intersect, the nodes containing each entry must also intersect. Therefore, the algorithm recursively descends down each intersecting pair of nodes, starting at the roots until it reaches the leaves.

The work of Qi et al. [17] forms the algorithmic basis of this work. Their contributions are three algorithms for top-k spatial joins called the Distance-First Algorithm (DFA), the Score-First Algorithm (SFA) and the Block-based Algorithm (BA). Each algorithm focuses differently on the score attributes and spatial attributes of the input objects. The DFA and BA use a ranked version of the R-tree spatial intersection join. SFA and BA borrow aspects of previous implementations of top-k queries and top-k joins by reading input items in descending order of rank. BA is found perform best in general. The algorithms are not explicitly designed to be multithreaded, nor is the use of GPUs to implement the algorithms considered.

You et al. [21] and Luo et al. [14] achieved significant speedups for parallel evaluation of multiple spatial queries on R-trees on GPUs. Their works describe two alternative solutions for memory representation of R-trees and bulk loading of R-trees on the GPU. [21] describes the linearized R-tree layout, which represents the R-tree as a single compact array of nodes that can easily be streamed between CPU and GPU, and uses an algorithm called Sort-Tile-Recursive to bulk load R-trees by using parallel sorting and reduction primitives implemented on the GPU. [14] describes a similar layout based on two arrays, one containing the MBBs and the other containing node references as well as data entries, which is bulk loaded using a much cheaper algorithm. The experiments performed by You et al. suggest that their more expensive bulk loading algorithms produces R-trees with better parallel query performance. In their experiments, they perform spatial queries in parallel using basic graph traversal methods and find that their implementation of breadth-first search generally outperforms depth-first search on a GPU. They suggest a number of strategies for dealing with the limited capacity of shared memory for queues in breadth-first search.

Yampaka & Chongstitvatana [20] describe methods to perform spatial join operations on R-trees on a GPU. While their methods are somewhat unclear, they demonstrate a clear possibility of parallelization of the R-tree join using concurrent tree traversal as described by Brinkhoff el al. Given two R-tree nodes to be joined, the GPU can efficiently assign one lightweight thread to each pair of R-tree node records to parallelize the evaluation of the spatial predicate for each pair of R-tree node records.

Bouros et al. [4] review research and recent trends in the evaluation of spatial intersection joins and suggest future directions for research. Many techniques were designed for disk resident data, but the recent trend of memories becoming much larger allows more of the data to reside in memory, which calls for techniques designed for in-memory processing. Another trend is the use of distributed and parallel processing, where methods to process spatial joins using the MapReduce paradigm are reviewed. They suggest that one direction for future research should be “scaling up” into shared-memory multi-core systems or using GPUs instead of “scaling out” into distributed systems because common datasets can fit in the memory of commodity machines. They also suggest that extended join operations require more research, which are spatial join operations which deal with more than spatial attributes. One such extended join operation would be the top-k spatial join.

Chapter 2

GPGPU

General-purpose computing on graphics processors (GPGPU) describes the use of a graphics processing unit (GPU) to perform computation that would traditionally be performed on the central processing unit (CPU). The majority of GPUs today are programmable using APIs such as OpenCL and CUDA. Using these APIs, applications today can utilize GPUs as coprocessors to execute some parts of an application that are especially suited for GPUs. GPGPU is a much larger topic than what this chapter can cover, therefore the chapter focuses on the GPGPU topics that are relevant for the implementation of top-k spatial joins on GPU.

GPUs are particularly well-suited for problems that can be expressed as data-parallel computations, which is executing the same program on many data items in parallel, ideally with high arithmetic intensity. Arithmetic intensity is the ratio of arithmetic operations to memory operations. Graphics processing is the original application, but GPUs have found widespread application in other areas such as machine learning, where popular tools achieve significant speedups by moving most of the computation to the GPU.

This chapter focuses on the architecture of NVIDIA GPUs and the CUDA API [6]. The alternative would be OpenCL, which is an open API available on hardware from other vendors, which can also run on other types of hardware. CUDA was chosen for this thesis due to its widespread use and the quality of its documentation and tooling. Additionally, the relation between the design of the CUDA API and the graphics hardware yields additional insight. OpenCL is designed for portability, so that it can be executed on GPUs from the vendors who support (even CPUs), but the portability is not a major concern for this thesis.

An overview of relevant aspects of GPGPU and CUDA was created in the project preceding this thesis [1]. This chapter is a revised version of that overview, with some clarification, corrections and amendments of additional topics that became relevant for the actual implementation of top-k spatial joins on GPU.

2.1 GPU architecture

GPUs have some significant architectural differences to CPUs, something programmers must be aware of to fully utilize the capabilities of GPUs. The architecture of CPUs is designed to perform well for a variety of applications, while the architecture of GPUs is designed primarily for graphics applications. The main differences are in the execution of threads and their memory models. A CPU typically operates on one large random access memory, while GPU memory is divided into multiple regions with different capacities, bandwidths, latencies and access methods. CPUs often have a small amount of cores for parallel execution of threads that can communicate via memory or interrupts, while threads on GPUs are created in bulk and can communicate in a number of specialized ways.

GPU architectures have evolved over time due to technological advancements and evolving needs of graphics applications, which exposes new features to programs and influences the way programs must be designed to optimally utilize the available resources. Which features of an NVIDIA GPU device are identified as the *compute capability*, which is a string consisting of a major and minor version number, which at the time of writing ranges from 1.x to 7.x. This chapter describes devices with compute capability 6.x and 7.x, which is available in the most recent generations of hardware at the time of writing.

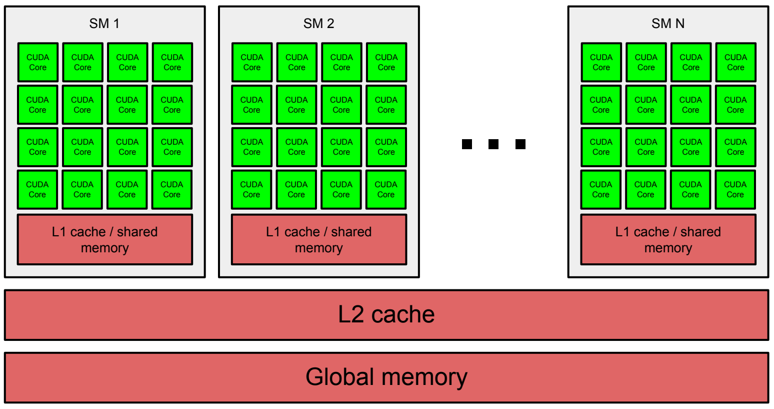

Figure 2.1: Simplified overview of GPU architecture

Unlike the CPU, the GPU dedicates most of its resources to data processing instead of caches and control flow mechanisms. Therefore, the cost of memory accesses and branching is high compared to the cost of data processing such as arithmetic operations. This means that programs with high arithmetic intensity are favored. The individual performance of each thread on a GPU is much lower than the performance of a thread on a CPU, but the GPU makes up for it with the sheer amount of threads that it can execute in parallel.

A GPU contains an array of Streaming Multiprocessors (SMs) and a large shared global memory. The multiprocessors can access the global memory and have a shared L2 cache. Each multiprocessor has its own L1 cache and its own on-chip memory as well as a large amount of registers. The GPU and CPU are connected through an interface such as PCI-e. The bandwidth of the global memory from the multiprocessors is much higher than the bandwidth between the GPU and CPU. The bandwidth of a multiprocessor's on-chip memory is also much higher than the bandwidth to the global memory.

2.1.1 SIMT architecture

Work is carried out by creating groups of threads to run a program, and distributing threads among the multiprocessors in groups of 32 parallel threads called *warps*. A group of warps may have interdependent threads and must be scheduled onto the same multiprocessor. Each multiprocessor creates, manages, schedules and executes threads in warps. A warp is the smallest unit of execution, meaning that instantiating a number of threads that is not divisible into warps requires leaving a remainder of unutilized threads. All threads in a warp share a single program but have their own program counters and registers. Warps execute independently, but may share a program and shared memory with other warps residing on the same multiprocessor, and may also share a program and global memory with warps on other multiprocessors.

Within the multiprocessors, warps are executed with a *Single Instruction, Multiple Thread* (SIMT) architecture, meaning a warp executes one common instruction at a time on all 32 threads in parallel. Full efficiency is realized when all 32 threads have non-divergent execution paths. All participating threads will typically run in sync, but execution paths may diverge when there are data-dependent jump instructions that cause threads to jump to different parts of the program. When the execution paths diverge, the warp can execute only the common instruction of one subgroup of threads at a time by temporarily disabling threads that are not part of the subgroup.

The execution of warp instructions within a multiprocessor is interleaved and there is no context switching cost when executing instructions from different warps. Interleaved execution means that instructions that the warp scheduler deem independent may be executed concurrently, so that a new instruction can be executed while the results of other independent instructions are still pending. This interleaving allows both concurrent execution of instructions from the same program within the same warp, as well as concurrent execution of instructions across multiple warps. Unlike a CPU core switching between threads, a context switch from one warp to another in a multiprocessor has no cost, which is a major part of the extremely lightweight nature of the threads. This is made possible by maintaining the program counters, registers and shared memory used by a warp on-chip during the entire lifetime of the warp, even while other warps are executing. Each multiprocessor has a fixed amount of registers that is allocated between the warps and a limited shared memory. The amount of warps that can be executed concurrently by one multiprocessor is therefore limited by the amount of registers required by each warp and how much shared memory they require.

2.1.2 Memory

Each multiprocessor has on-chip memory with higher bandwidth and much lower latency compared to that of device memory. The local memory is a sequence of 32-bit words distributed into 32 equally sized memory banks. Each memory bank has a bandwidth of 32 bits per clock cycle. A shared memory request for a warp allows each thread to access its own 32-bit word in a single clock cycle provided there is no *bank conflict*. A bank conflict occurs when multiple different words residing in the same memory bank are being accessed concurrently. A memory bank can broadcast only a single 32-bit word at a time, so concurrent access to multiple words within the same bank cannot be serviced in a single clock cycle. If a bank conflict occurs, the request is split into as many separate conflict-free requests as necessary.

Device memory is accessed via naturally aligned 32-, 64- or 128-byte memory transactions with an important feature called *coalescence*. When a warp accesses device memory, each thread will typically try to access its own address in device memory. Instead of servicing each request individually, the warp will attempt to coalesce concurrent device memory access into as few memory transactions as possible. Optimal throughput is generally achieved with the smallest amount of transactions with the least amount of unused words. Therefore, an efficient program must exhibit dense device memory access patterns so that warps access the same regions of memory. This often means that all the threads in a warp will access consecutive words from device memory. Additionally, the access patterns should be aligned with the memory transactions.

2.2 CUDA

CUDA distinguishes two entities, the host and the device. The host has the main memory and the CPU that runs the host program, while the device is a GPU serving as a coprocessor with its own memory that can run its own programs. The host can allocate memory on the device, initiate memory transfers between host memory and device memory, and can schedule programs to run on the device.

The most popular way to use CUDA is by extensions to C or C++ allowing to write programs with parts that run on the host and parts that run on the device, even with parts that can run on both the host and the device. The parts of the program that can run on the device are written using the same programming language syntax but use special annotations to utilize parts of the CUDA API, creating a low barrier of entry for C/C++ developers. Feature parity with C and C++ for code that runs on the device is mostly preserved, but there are some notable documented exceptions.

2.2.1 Execution model

The unit of execution in CUDA is a kernel invocation, which consists of a number of extremely lightweight threads divided into thread groups. A kernel is a program written as a function that can be executed N times in parallel by N different CUDA threads. When a kernel is invoked, the caller must specify the number of thread groups and the number of threads per group that will be used to carry out the execution. The division into thread groups forms boundaries for thread cooperation. Each thread group is a mostly independent part of the execution of the kernel, while the threads within each thread group can cooperate tightly.

CUDA provides a number of thread cooperation mechanisms both for thread groups and warps. A thread group has a shared memory that all participating threads may cooperatively operate on. Barriers may also be used to synchronize the execution of all threads in a warp. The grouping of threads into warps is also exposed to the programmer with special warp-level cooperation mechanisms.

Threads and thread groups are composed into grids. A kernel invocation has a grid of thread groups, and each thread group has a grid of threads. Grids are three-dimensional, but not all dimensions of the grid have to be used, so each item in a grid can be identified using a one-dimensional, two-dimensional or three-dimensional index. Each thread knows its own thread index and group index. The thread index and group index can be used to assign each thread or group to an individual item in a domain such as a vector, matrix, or volume.

Thread groups must be able to execute independently, because the order in which thread groups are executed and their degree of concurrency is unspecified. This means that cooperation between thread groups is limited. This programming limitation is key to allow flexibility for the device to find the optimal execution plan to schedule thread groups across multiprocessors based on its architecture and available resources.

All threads in a thread group are required to be scheduled onto the same multiprocessor. The threads are divided into $\lceil \frac{T}{W_{size}} \rceil$ warps where T is the number of threads per group and W_{size} is the warp size, equal to 32. All warps in a thread group are scheduled onto the same multiprocessor. The execution of each warp is independent and may be interleaved. Because all threads in a warp execute synchronously, the threads in a warp can depend on each others' writes to shared memory, using synchronization and memory fences to ensure they observe each others' writes. If the execution of a warp does not depend on the execution of other warps, it does not have to explicitly synchronize with other warps from the same thread group at all. Otherwise, barriers can be used for inter-warp synchronization.

CUDA supports useful special functions such as *warp votes* and *warp shuffles*. The SIMT architecture allows these functions to synchronize and exchange information between all the threads in the warp participating in the same function call. Shuffles allow threads to exchange variables with each other. A *shuffle up*, for instance, has each thread submitting a variable and reading the variable from another thread before it. Similarly, a *shuffle down* has each thread submitting a variable and reading a variable from another thread after it. Warp votes allow each thread to submit a predicate, returning a reduction of each predicate of participating threads, such as *any* which returns true if any predicate of each participating threads is true. A *ballot vote* returns a 32-bit integer whose N^{th} bit is set if the predicate of the N^{th} thread is true. Coupled with bitwise operators, ballots can be especially effective for certain kinds of warp communication.

In addition to the host launching kernels on the device, a kernel can also launch other kernels with a feature called *dynamic parallelism*. This is a particularly useful feature that allows applications with varying levels of parallelism to control more of the execution on the device. A kernel that uses dynamic parallelism can control the execution of other kernels with direct access to device memory, unlike the host which requires memory transfers and synchronization that can negatively affect the performance of the application.

2.2.2 Memory hierarchy

Memory in CUDA is divided into a hierarchy. Each CUDA thread has its own private memory, each thread group has its own shared memory, and all threads have access to a shared global memory. The hierarchy is not only a logical separation of memories that defines which threads have access to what — variables in different parts of the memory hierarchy are assigned to different physical memories on the GPU with different access characteristics.

Global memory is device memory that can be accessed by kernels. It is the memory with the greatest capacity, typically measured in gigabytes, but is also the memory with the greatest access cost. Both the host program and kernels can dynamically allocate global memory, and the host can initiate memory transfers between host memory and global device memory. Global memory is stored in device memory (VRAM) and is cached by L1 and L2 caches.

Shared memory is local to each thread group. It has much lower access cost than global memory, but its capacity is limited by the capacity of the on-chip memory, which requires using less than around 48 KB or 92 KB depending on the hardware. Shared memory is allocated semi-statically — it is allocated only once a thread group is assigned to a multiprocessor, where the amount of memory allocated to each thread group is specified or determined statically when invoking a kernel. Unlike global memory, additional shared memory cannot be allocated during kernel execution. Shared memory is stored in the on-chip memory banks, and should be accessed in ways that avoid bank conflicts.

Private memory is memory that belongs to each thread, which has the lowest potential access cost. It contains the local variables used by a kernel function. Variables in private memory are primarily stored in statically allocated multiprocessor registers. In some cases, private memory may have to be stored in what is (confusingly) called local memory, which is thread-local device memory. *Register spilling* may happen if each thread uses too much private memory to fit into the multiprocessor registers, causing CUDA to place some private memory in local memory. Because registers are not addressable like global or shared memory, CUDA may sometimes also choose to use local memory to make private memory addressable. Because local memory resides in device memory, it should generally be avoided to avoid the same latencies associated with accessing global memory. However, local memory is organized so that consecutive 32-bit words are accessed by consecutive threads. Thus, the cost of accessing local memory can be reduced if each thread in a warp accesses the same variable in local memory, causing the access to be fully coalesced.

Both global and shared memory support atomic operations. Atomic operations are another way for threads to safely cooperate by doing read and write operations without interference from concurrent execution. Global memory atomic operations are particularly useful for intra-thread group communication. For instance, if each thread group will write one item at a time to a shared buffer in global memory, a counter in global memory can be used with atomic addition operations to ensure the writes to the buffer are consistent.

2.2.3 Performance optimization

The performance of a CUDA application can scale with hardware advancements in multiple ways. The addition of more multiprocessors and increased parallelism within each multiprocessor can increase performance, provided the application can efficiently utilize all multiprocessors, usually by scheduling a sufficient amount of thread groups. New features added with increased compute capability can be utilized. The clock speed, instruction speed and memory sizes and bandwidths can all be increased. CUDA kernels are usually compiled into a portable intermediary instruction format called PTX which allows the driver for a particular piece of hardware to compile it further to optimize for architectural features.

A program should create as many work efficient independent parts of parallel execution as reasonably possible while respecting the warp size. A program that parallelizes over a set of threads with little cooperation between threads should generally prefer dividing the set of threads into as many independent thread groups as possible with a minimum size of 32 threads each. This gives the GPU freedom in how to schedule the thread groups across multiprocessors, and also allows interleaved execution within each multiprocessor. The amount of thread groups that can be assigned to each multiprocessor depends on the shared memory requirements of each thread group as well as the amount of registers utilized by each thread. Therefore, programs should also strive to minimize their memory requirements.

Programs should optimize for efficient memory access. Because shared memory is faster than global memory, programs should utilize registers and shared memory to cache frequently accessed global memory data that may be shared by threads in thread groups. Naturally, applications should also strive to avoid bank conflicts in shared memory with special care for shared memory access patterns. Requests to device memory should optimize for coalescence. This is achieved by making requests to dense areas of memory and accessing memory with naturally aligned 32-, 64- and 128-byte transactions. Otherwise, some memory bandwidth will be wasted on reading and writing extra bytes, caching becomes less effective, and excessive memory transactions will have to be performed. A common method to achieve good memory access patterns is by adding padding to data structures that are stored in arrays. For instance, a structure consisting of three single-precision floating point numbers (4 bytes each) should often be padded with 4 bytes so that the full data structure is aligned with 16 bytes.

Chapter 3

Spatial indexing

The goal of indexing is to improve the efficiency of searches. For spatial indexing specifically, the goal is to improve the efficiency of processing spatial queries. Spatial queries relate to the attributes of spatial objects such as their positions and sizes and whether or not they intersect with each other. The R-tree is a commonly used data structure for spatial indexing that can be used to accelerate spatial queries.

Research into spatial indexing was carried out in the project preceding this thesis [1]. This chapter is a revised edition of the spatial indexing chapter from that project.

3.1 Concepts

A spatial object can be a point, a line, a polygon, or any other shape in the real coordinate space of d dimensions $\mathbb{R}^d$. Using set theory, a spatial object in $\mathbb{R}^d$ can be considered as a potentially infinite set of points X such that $X \subseteq \mathbb{R}^d$.

Set operations have geometric interpretations. As an example, given two spatial objects X and Y, $X \cap Y$ is a spatial object consisting of the space covered by both objects, which is the intersection of the two objects.

A spatial predicate is a relation between spatial objects. For instance, given spatial objects U and V, U and V are said to intersect if $U \cap V \neq \emptyset$, and U is said to contain V if $U \supseteq V$. Spatial predicates may have properties that are useful for processing spatial queries. For instance, any object that contains an object U that intersects with an object V must also intersect with V.

A point p in d-dimensional space can be represented as a d-dimensional vector $(p^{[1]}, p^{[2]}, \ldots, p^{[d]})$ where $p^{[i]}$ denotes the i^{th} coordinate of p. A point can be considered as a spatial object in the form of a set with exactly one item, the point itself.

An axis aligned box is a spatial object in the form of a box where the edges of the box are parallel to the coordinate axes of the space. An axis aligned box is defined by two points $B = (b, t)$ where

$$\forall i \in \{1, \ldots, d\} : b^{[i]} \leq t^{[i]}$$

where $p^{[i]}$ denotes the i^{th} coordinate of the point p. b can be considered the *bottom point* and t can be considered the *top point*. The axis aligned box contains points so that for each dimension i, $b^{[i]}$ is the minimum (bottom) coordinate and $t^{[i]}$ is the maximum (top) coordinate. Thus, the set of points in the axis aligned box, defining it as a spatial object, can be expressed as the following:

$$B = \{q \in \mathbb{R}^d \mid \forall i \in \{1, \ldots, d\} : b^{[i]} \leq q^{[i]} \leq t^{[i]}\}$$

Axis aligned Minimum Bounding Boxes (MBB) are often used to approximate the shape and location of more complex spatial objects. The axis aligned MBB of a spatial object X is the axis aligned box with the smallest *measure* that covers X. Depending on the dimensionality of the space, the measure of a spatial object can be the length, area, volume or hypervolume of the object. In other words, an axis aligned MBB is the smallest possible axis aligned box that contains X.

An important corollary is that if the MBB of X covers X, it follows that any spatial object that intersects with X must also intersect with the MBB of X. Because testing spatial predicates on axis boxes is computationally cheap, MBBs are often used to speed up the evaluation of spatial predicates for more complex spatial objects.

The axis aligned MBB of a spatial object X can be represented as $B = (b, t)$ where

$$\forall i \in \{1, \ldots, d\} : \left[b^{[i]} = \min\left(q^{[i]} \mid q \in X\right), t^{[i]} = \max\left(q^{[i]} \mid q \in X\right)\right].$$

3.2 Spatial join queries

The spatial distance join requests pairs of objects that are within a certain distance from each other. More formally, given two sets of spatial objects R and S, a distance metric *dist* and a distance threshold ε, the spatial distance join returns

$$\{(r, s) \in R \times S \mid \mathrm{dist}(r, s) \leq \varepsilon\}.$$

dist is defined to return the distance between the points of the two spatial objects that are closest to each other according to a point distance function p.

$$dist(r, s) = \min\left(p(t, u) \mid t, u \in r \times s\right)$$

For instance, p could be the Euclidean distance, which is the straight-line distance between two vectors. We can also use the Chebyshev distance, where the distance is the greatest of the differences along any coordinate dimension.

$dist(r, s) \leq \varepsilon$ is a type of spatial predicate. For $\varepsilon = 0$ it is equivalent to intersection. Using the Chebyshev distance measure between two bounding boxes r and s, we can think of it as expanding either r or s by ε and testing for intersection.

For the purposes of spatial indexing, the distance metric will primarily operate on pairs of points and pairs of axis aligned boxes. This simplifies computing the pairs of points that are closest to each other according to the distance function. For pairs of spatial objects that are points, the distance function is simply computed directly on the points. For pairs of spatial objects that are axis aligned boxes, the closest points can be found on the box boundaries, unless the boxes intersect, in which case the distance is zero.

3.3 R-trees

The R-tree [8] can be considered a multidimensional version of the B-tree, which is a balanced search tree. Instead of using ordinal key ranges for searching like the B-tree, the R-tree uses axis aligned bounding boxes which can be spatially queried.

The R-tree supports inserting and deleting objects as well as searching for objects by their spatial properties. Its properties makes it a highly useful indexing data structure for spatial queries.

Conceptually, the R-tree partitions the space into a hierarchy of MBBs where the MBBs of the objects in the R-tree are placed at the bottom. Each MBB in the hierarchy consists of the MBB of its children. MBBs at the same level in the hierarchy are allowed to overlap with each other, but should overlap as little as possible.

The R-tree is a balanced tree structure consisting of nodes divided into levels. Nodes in all levels but the lowest level are called inner nodes, while nodes at the lowest level are called leaf nodes. Objects in the R-tree are placed as entries in the leaf nodes, each containing a link to a data object along with the MBB of the object. Inner nodes contain similar entries, except they link to lower level nodes instead of data objects along with the MBB of the lower level node. A leaf node is said to contain a data object if any of its entries point to the data object. Similarly, an inner node is said to contain a data object if any of its ancestor leaf nodes contain the data object. In figure 3.1, A is the root node, which is an inner node. B, C and D are leaf nodes which are entries of the root node. E, F, G, H, I and J are leaf node entries.

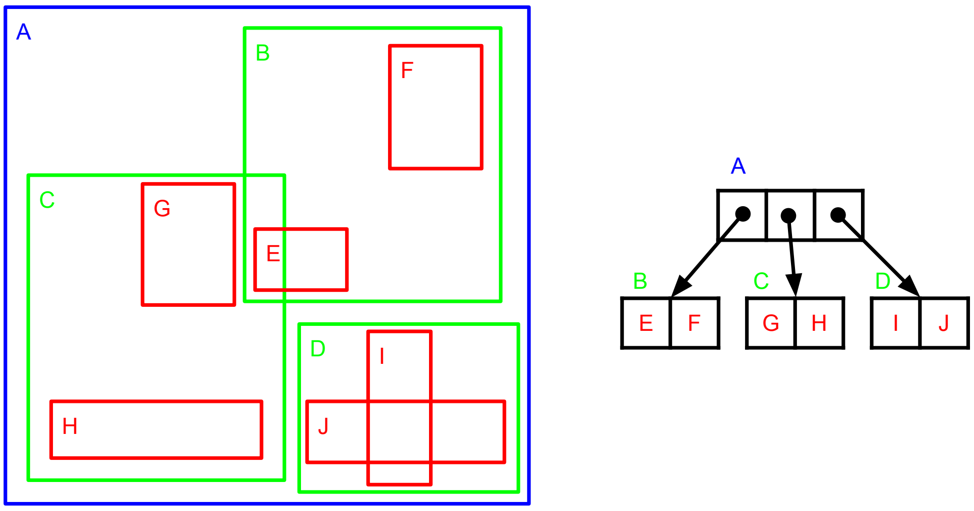

Figure 3.1: R-tree

To ensure that the R-tree remains balanced, there are restrictions for the minimum and maximum amount of entries that a node may contain. All nodes but the root node must have between m and r entries where $m \leq \frac{r}{2}$. r is the maximum amount of entries in an R-tree node, and is called the *fanout*. The root node can have anywhere from 0 to r entries. A node that has r entries is considered *fully packed*.

To insert an object into the R-tree, it must be inserted into an appropriate leaf node. If inserting an entry into the leaf node would result in more than r entries, the node has to be split into two leaf nodes by creating a new leaf node and partitioning the entries between them. The newly created leaf node has to be inserted into the parent node, which may result in the parent node requiring a similar split. The split can propagate up to the root, in which case the root also must be split, and another level has to be added to the tree.

The query performance of an R-tree is a measure of the average amount of nodes that have to be visited to perform a search. When searching it is desirable to visit as few nodes as possible, as each branch of each node has to be evaluated, and accessing a node may have a cost (depending on the memory layout). For good query performance, R-tree nodes should be packed as fully as possible, and branches should have minimal overlap. Packed nodes decrease the height of the tree, which reduces the amount of nodes that have to be accessed to reach the leaves. When the cost of accessing a node outweighs the cost of evaluating branches, packed nodes reduce the total amount of nodes which results in fewer nodes that have to be accessed to perform a search. Minimal overlap between branches allows the search to more easily narrow down to fewer subtrees.

The R-tree has been extensively researched and used, and has inspired the creation of variants such as the R*-tree [3], the R+-tree and the revised R-tree. Most of the variants preserve the data structure and principles while redefining the insert and remove operations. Operations that modify the tree can be designed for varying degrees of query performance at the cost of the performance and complexity of operations to insert and delete values.

3.3.1 Range search using R-trees

A range search on an R-tree returns the leaf entries whose MBBs intersect with the query box. Algorithm 3.1 can be described as an iterative operation on a LIFO (Last-In-First-Out) queue. The queue is initialized with the entries of the root node. Entries are dequeued until the queue is empty. If the dequeued entry is a leaf node entry, it is made part of the result. If the entry is an inner node entry, all its children are enqueued. A query box that does not intersect with the MBB of a node N cannot intersect with any objects contained by N. Therefore, a non-intersecting node and all its ancestors can be pruned from the search because they will not output any results.

While range search concerns spatial intersection with a query box, it is possible to generalize for other predicates provided they have a similar property. The specific property required of a predicate ϕ is as follows,

$$\forall A, A' \in \mathbb{P}(\mathbb{R}^d) : (A \subseteq A') \land \phi(A) \Rightarrow \phi(A')$$

where $\mathbb{P}(\mathbb{R}^d)$ is the set of all spatial objects in the d-dimensional space $\mathbb{R}^d$. An interpretation is that a if the predicate is true for a spatial object A, it must also be true for any larger object that contains A. In the case of an R-tree range query, the spatial predicate ϕ tests for intersection with a query box, where A is a box of a leaf node entry and A' is the MBB of any node that contains it. If an entry in an R-tree intersects with the query box, any node that contains the entry must also intersect with the query box, otherwise it would be pruned during the range search.

3.3.2 Spatial join using R-trees

R-trees can naturally be used to enhance the performance of spatial joins. As opposed to performing a nested loop join on lists of spatial objects to test the spatial

Algorithm 3.1 R-tree Range Search. N is any R-tree node, but usually the root node of the R-tree. Q is the query box.

```
 1: function RangeSearch($N, Q$)
 2:     $O \leftarrow \emptyset$
 3:     Initialize $Queue$ as a queue with entries of $N$.
 4:     while $Queue$ is not empty do
 5:         $E \leftarrow$ Dequeue($Queue$)
 6:         if Intersects($E.mbb, Q$) then
 7:             if $E$ is a leaf entry then
 8:                 $O \leftarrow O + E$
 9:             else
10:                 for all $C \in E.ref$ do
11:                     Enqueue($Queue, C$)
12:                 end for
13:             end if
14:         end if
15:     end while
16:     return $O$
17: end function
```

predicate for all pairs, R-trees can utilize the spatial characteristics of the inputs to efficiently navigate the search space and reduce the amount of work required to produce the output. R-trees can be built dynamically, similarly to how hash tables are used for relational joins, or they can be joined directly.

Brinkhoff et al. [5] described a method to concurrently descend down two R-trees to perform a spatial intersection join as described in algorithm 3.2. The algorithm is quite similar to the range search algorithm, except it operates on pairs of node entries and searches for leaf entry pairs instead of individual leaf entries. Similar to the pruning in the range search algorithm, the spatial intersection join is based on the idea that if the MBBs of nodes N_R and N_S do not intersect, the objects they contain cannot be joined into any pair of intersecting objects. Given well constructed R-trees, this allows pruning many pairs of objects at directory levels in the R-tree hierarchy, saving a lot of work.

Like the R-tree range search algorithm, the spatial intersection join algorithm can be described as an iterative operation on a LIFO (Last-In-First-Out) queue. The queue is initialized with all root node entry combinations. Tuples of entries are dequeued until the queue is empty. If the dequeued entries are leaf node entries, they are made part of the result. If the entries are inner node entries, all combinations of children entries are enqueued. A pair of nodes that do not intersect cannot contain intersecting entries. Therefore, tuples of non-intersecting nodes and all their ancestor tuples can be pruned from the search because they will not output any results.

For R-trees with different heights, a more complete solution is required. The algorithm can only descend as deep as the least tall R-tree until it reaches leaf node entries. In the case where S is taller than R, the spatial intersection join algorithm will output entries (E_R, E_S) where E_R is a leaf node entry in R while E_S is an inner node entry in S instead. One method to produce the expected spatial join outputs here would be to carry out a range search for each output of the spatial intersection join algorithm. For a pair of entries (E_R, E_S), the node referenced by E_S is searched

Algorithm 3.2 R-tree Spatial Intersection Join. N_R and N_S are R-tree nodes from R-trees R and S, usually the root nodes of their respective R-trees.

```
function SpatialJoin($N_R, N_S$)    ▷ $N_R$ and $N_s$ must be at the same level
    $O \leftarrow \emptyset$
    Initialize $Queue$ as a queue with entry pairs of $N_R \times N_S$.
    while $Queue$ is not empty do
        $(E_R, E_S) \leftarrow$ Dequeue($Queue$)
        if Intersects($E_R.mbb, E_S.mbb$) then
            if $E_R$ and $E_S$ are leaf entries then
                $O \leftarrow O + (E_R, E_S)$
            else
                for all $C_R \in E_R.ref$ do
                    for all $C_S \in E_S.ref$ do
                        Enqueue($Queue, (C_R, C_S)$)
                    end for
                end for
            end if
        end if
    end while
    return $O$
end function
```

using E_R.mbb as the query box. The output of each range search on E_S can then be output as pairs with E_R.

While Brinkhoff et al. only considered spatial intersection, it is possible to generalize for other spatial predicates provided they have a property similar to the property that generalizes the range search. The specific property required of a spatial predicate ϕ used to join the objects is as follows,

$$\forall A, A', B, B' \in \mathbb{P}(\mathbb{R}^d) : (A \subseteq A') \wedge (B \subseteq B') \wedge \phi(A, B) \Rightarrow \phi(A', B')$$

where $\mathbb{P}(\mathbb{R}^d)$ is the set of all spatial objects in the d-dimensional space $\mathbb{R}^d$. An interpretation is that if the predicate is true for a pair of spatial objects A and B, it must also be true for any pair of larger objects that contain A and B. In the case of an R-tree join, A and B can be considered as entries from their respective R-trees, and A' and B' are the MBBs of nodes containing them. If entries of two R-trees intersect with each other, any node that contains the entries from each tree must also intersect with each other, otherwise they would be pruned during the range search.

The generalization of spatial predicates for range search and spatial joins is a necessary insight to apply the algorithms for other predicates than intersection. The property is intuitively true for the intersection predicate. The spatial distance threshold predicate can be considered equivalent to intersection where either object is *expanded* by ϵ, therefore the property intuitively holds true for the spatial distance predicate.

Queues can be implemented in various ways for range searches with R-trees and R-tree joins to perform the search or join in different orders. By replacing the Last-In-First-Out (LIFO) queue with a First-In-First-Out (FIFO) queue, the search becomes a breadth-first search (BFS) instead of a depth-first search (DFS).

Another alternative is using a priority queue to perform a best-first search, which has applications for ranked queries.

3.3.3 R-tree search methods

A depth-first-search can be performed using a compact tree search iterator such as the one described in algorithm 3.3 instead of using an explicit queue. The iterator operates on a stack of node entry iterators. The stack will contain at most an iterator per level of the R-tree, and therefore has a fixed capacity equal to the height of the R-tree. The node entry iterator may simply be based on a pointer to an R-tree node and a number that counts the number of entries of the node that have been visited. The tree search iterator can also be applied to spatial joins by replacing the node entry iterator with an iterator that joins the entries of two nodes.

Algorithm 3.3 DFS Tree Search Iterator. S is a stack of node entry iterators, initialized with a node entry iterator of the root node. ϕ is the search predicate used to prune R-tree entries.

```
1:  function Next($S, \phi$)
2:      while $S$ is not empty do
3:          $I \leftarrow$ Peek($S$)
4:          $E \leftarrow$ Next($I, \phi$)
5:          if $E$ is Nil then
6:              Pop($S$)
7:          else
8:              if $E$ is inner node entry then
9:                  $I_C \leftarrow$ NodeEntryIterator($E$)
10:                 Push($S, I_C$)
11:             else
12:                 return $E$
13:             end if
14:         end if
15:     end while
16:     return Nil
17: end function
```

An explicit queue based search opens the possibility for parallelism by having multiple threads dequeuing, processing and enqueuing items in the same queue. This method can achieve data parallelism by both evaluating spatial predicates and traversing subtrees in parallel. The efficiency of this method depends on the amount of items that can be processed in parallel, which may be limited by the amount of queue items produced at each level in the search, as well as the overhead of synchronizing the queue. For a narrow range search in a well constructed R-tree, the number of subtrees that must be visited should be low, which results in an overall small queue until the last few levels. Therefore, the size of the queue may limit the parallelism of the algorithm. However, for an R-tree spatial join, the queue is expected to grow exponentially, which suggests that the parallelism of spatial joins may be limited by the amount threads available rather than the size of the queue.

One way to utilize parallel processing of queue items in range search is processing multiple range queries in parallel using the same queue. You et al. [21] proposed and evaluated multiple methods to perform parallel range queries using GPUs. Given an R-tree and a set of range queries to be performed on the R-tree, the algorithms efficiently computes the result of all range queries in parallel. It can be done by implementing existing methods for depth-first and breadth-first search for R-tree range queries on the GPU.

In the DFS method, each thread processes one query in depth-first order using a DFS tree search iterator. The main advantage of DFS is its predictable memory usage and simplicity. Each query requires only a stack with capacity equal to the height of the tree to perform the traversal, which can be stored in shared memory. The implementation is based on a count-and-write strategy that spaces out the memory prior to writing the query results. An issue with the DFS method is that it provides poor workload balance when queries require different amounts of work, leaving some threads idle and waiting for others to finish. The count-and-write strategy effectively requires evaluating all queries twice, which is speculated to harm performance. For particularly large queries, the writing of results to global memory may also be too sparse to coalesce.

The BFS method distributes the queries across multiple thread groups where each thread group has a queue in shared memory and all its threads working on the queue in parallel. Each queue entry is a (N, Q) tuple where N is a pointer to an R-tree node and Q identifies a query. A queue entry (N, Q) is expanded by dequeuing then enqueueing the children of N as (N_C, Q) tuples, but only if the MBB of N and the query box of Q intersect. When a queue contains only leaf nodes, it is copied to global GPU memory to be made part of the result. Compared to the DFS method, the BFS method solves the workload balance problem within each thread group and has superior performance overall, but is more complex due to the limited capacity of the queue.

As previously stated, each queue in the BFS method has a fixed capacity imposed by the limited amount of shared memory that is available to each thread group. In a breadth-first search, the queue is expected to grow larger as the search goes deeper and may cause the queue to overflow. Calculating a distribution of queries that avoids overflow is infeasible, therefore the algorithm is designed to work correctly even when the queue in shared memory overflows. There are multiple alternatives for overflow handling strategies, and the most promising alternative dynamically allocates chunks of global memory for excess queue entries.

There is an equivalency between performing a set of range queries on an R-tree and performing an R-tree spatial intersection join. By creating another R-tree from the set of range queries and performing a spatial intersection join, the same result should be produced. By decomposing an R-tree into its objects, a spatial intersection join can be evaluated by performing a range query for each object. Therefore, spatial join methods could be applied to perform range queries in bulk. Conversely, the methods for performing parallel range queries could be applied to perform parallel spatial joins. Spatial join methods are better suited to exploit the spatial properties of the set of range queries, but may fall short when the amount of queries is low.

Given the similar queue-based search nature of DFS, BFS and the R-tree spatial join algorithm, it may be possible to adapt the aforementioned DFS and BFS methods to perform parallel spatial joins instead of parallel range queries, but there are some challenges. A key observation is that the parallelism is limited by the amount of entries in the queue. Execution of parallel range queries like in DFS and BFS starts with a large queue containing each initial query, and therefore exhibits a high degree of initial parallelism. A spatial join, however, starts with a single queue entry and therefore exhibits no initial parallelism and would have a slower start.

3.3.4 Memory layout

Being originally designed for disk storage, the original R-tree uses a page-based layout. Nodes correspond to disk pages if the structure is disk resident, and memory pages otherwise. An R-tree node with this layout can contain as many entries as the page can fit as records. A notable feature is that the MBB of an R-tree node is stored along with the pointer to the node's page in an inner node record instead of being stored within the node's own page. This allows pruning the node without having to load the page. Loading a disk resident page has a significant I/O cost, but a memory resident page may also have an access cost because the page may have to be loaded from memory into a cache. The main advantage of the page-based layout is that modifying the structure of the R-tree only requires allocating and freeing pages and updating pointers to pages.

A page based layout would be inefficient on GPU. One difficulty with using the page-based layout on GPUs is transferring R-trees between main memory and device memory. Every source page would have to be replicated in the other memory by allocating a destination page and performing a memory transfer between the source and the destination per page. Unless a unified virtual address space is used, destination pages may receive different memory addresses in the destination memory, requiring pointers to be updated as well. Additionally, R-trees with read-only usage rarely benefit from the advantages of the page-based layout, which is its performance for operations that modify the structure of the R-tree.

Instead of a page-based layout, Luo et al. [14] used two arrays to store the R-tree structure on a GPU. The first is an array of integers representing the tree structure which is called *Index*, and the second is an array of boxes called *Rect*. For an R-tree with N nodes and a fanout of r, both arrays consist of N blocks of r values. The root node is stored in the first block of both arrays. For inner nodes, each integer in a block of *Index* is the array index of a child node, and each box in a block of *Rect* is the MBB of the child node. For leaf nodes, each integer in a block of *Index* is an identifier of an entry in the R-tree, and each box in a block of *Rect* is the box of the entry. Non-full nodes are padded by zeroes in *Index*.

The linearized R-tree memory layout used by You et al. [21] stores the entire R-tree in an array of all nodes in breadth-first order. Inner nodes are represented by $\{MBB, pos, len\}$ tuples, where MBB is the minimum bounding box of the node, pos is the position of the first child node in the array, and len is the amount of children. Because sibling nodes are stored sequentially in BFS order, their positions can be calculated by offsetting from the position of the first child. The specific layout of leaf nodes was left unspecified. The linearized R-tree memory layout is more compact than the layout used by Luo et al. because inner nodes do not store a position per child. It does not pad nodes that are not filled to capacity, which may negatively affect memory reading patterns, but saves some memory.

The linearized R-tree memory layout is optimized for read operations and memory transfers without serialization. It is designed to be cache friendly with its compact layout and sequential sibling node storage. The entire R-tree can easily be streamed between host memory and device memory by only streaming the the array. It should only be used for read-only R-trees. Write operations would require shifting nodes around in the array and recalculating pointers to child nodes to preserve DFS order and thus would perform poorly. Linearized R-trees can really only be constructed when the node layout is known in advance. Bulk loading methods may be adapted to construct linearized R-trees, or alternatively, a linearized R-tree may be constructed as a copy of another R-tree.

3.3.5 Bulk loading

R-tree bulk loading is the process of constructing an R-tree from an unindexed set of object. When querying a set of unindexed objects, it may be beneficial to bulk load an R-tree prior to performing the queries, but bulk loading can be a costly operation. Prior bulk loading may enhance the performance of queries on unindexed sets, but only if the cost of performing unindexed queries outweighs the cost of bulk loading the R-tree and performing indexed queries. The choice of bulk loading strategy strongly affects the query performance. Bulk loading strategies often produce packed R-trees, which generally increases the query performance and reduces the memory usage of the R-tree.

Dynamic bulk loading is the most basic form of bulk loading where the R-tree is constructed by inserting one entry at a time. It is inherently sequential and relies on the efficiency of the insertion algorithm. A slow insertion algorithm may produce an R-tree with good query performance, but makes for a slow bulk loading strategy. A cheap insertion algorithm may make a faster bulk loading strategy, but the query performance of the resulting R-tree will suffer. Dynamic insertion also relies on dynamic allocation of nodes, which may be costly. Most insertion algorithms used for dynamic bulk loading will not produce packed R-trees, which limits the query performance.

Sort-Tile-Recursive (STR) is a bulk loading algorithm described by Leutenegger et al. [12] that recursively partitions the dataset into tiles with equal amounts of entries. Each recursion step relies on a series of passes to sort and split the data. When the layout of a linearized R-tree is known in advance, STR can be used for bulk loading linearized R-trees. In each iteration, STR splits a set of entries once per dimension — first all entries are divided into vertical slices by sorting by their x coordinate then dividing the series evenly, then each slice is split into tiles by sorting their entries by their y coordinate then once again dividing the series evenly. Each tile has at least r entries so that it can become an R-tree node. Then an inner node entry is produced for each tile to be processed in the next iteration. In each iteration, the amount of entries is roughly divided by r. When the amount of entries to be packed is lower than r, the root node is produced.

Chapter 4

Ranked queries

Ranked queries are queries concerned with the ranking of objects. With ranked queries, inputs and outputs may be objects that are ordered by a scoring function. A special case is the ranked join, which is a ranked query operating on the results of a join. In the case of the ranked join, we may use an aggregation of scores as the score of a pair of joined objects.

Research into ranked queries was carried out in the project preceding this thesis [1]. This chapter is a revised edition of the ranked queries chapter from that project.

4.1 Concepts

Given a set of objects R, a top-k query returns an ordered set of up to k objects of R with the greatest ranking. The ranking is defined by a scoring function q that assigns a score to each object in R. All items in the result set T must have a score greater than or equal to the score of the items that are not in the result set. More formally,

$$\forall_{r \in T} \forall_{s \in R-T} : (q(r) \geq q(s)) \,.$$

Given two sets of objects R and S, a top-k join returns an ordered set of up to k join results with the greatest ranking. It is a special case of the top-k query that operates on the results of a join. Join results are (r, s) tuples of $R \times S$ that satisfy a join predicate ϕ. The ranking is similarly defined by a score function q that assigns a score to each tuple of the join result. All items in the result must have a score greater than or equal to the score of the items that are not in the result set.

A score function assigns a score to each object in a set to define a ranking. The score may be an aggregation of multiple object score attributes $a_1, a_2, a_3, \ldots$ by an aggregate function γ. For instance, if γ is SUM, the score of an object is the sum of the score attributes of the object. γ is considered a monotone aggregate function if it is nondecreasing with respect to any parameter. More formally, γ is considered a monotone aggregate function if

$$\forall_{(x,y)} \forall_{(x',y')} : (x \leq x' \wedge y \leq y') \Rightarrow (\gamma(x, y) \leq \gamma(x', y')) \,.$$

SUM, MAX and MIN are all examples of monotone aggregate functions.

Using monotone aggregate functions for scores, some useful assertions can be made about the scores of sets. A key insight that is that if a set is scored by a monotone aggregation of attributes a_1 and a_2, the aggregation of the maximum value of a_1 and the maximum value of a_2 for all items forms an upper bound for the score of all items in the set. More formally, given a set R of items with attributes a_1 and a_2,

$$\forall r \in R : \gamma(r.a_1, r.a_2) \leq \gamma(\max(s.a_1 \mid s \in R), \max(r.a_2 \mid r \in R))$$

where $\max(s.a_1 \mid s \in R)$ and $\max(s.a_2 \mid\in R)$ are the maximum values of a_1 and a_2 for R. If it can be asserted that all items in a result must have a score greater than $s_{\min}$, then knowing the maximum value of each attribute for a set is enough to determine if any item in the set can be part of the result by comparing the upper bound with $s_{\min}$. If all items in the set cannot be part of the result, there is no need to evaluate each item for inclusion in the result by computing the score and comparing it with $s_{\min}$.

For top-k joins, the join inputs R and S may be individually ranked by score functions q_R and q_S respectively. If the scores assigned by q_R and q_S are considered as attributes of each joined tuple, then the score of a joined tuple can be an aggregation of the attributes assigned by q_R and q_S. In this case, the score of a joined tuple would be $q(r, s) = \gamma(q_R(r), q_S(s))$ where γ is an aggregate function.

For top-k joins where the score is a monotone aggregation of scores, the aggregation of the score of each highest ranked item in ranked sets R and S forms an upper bound for the score of all pairs in $R \times S$. More formally, given ranked sets R and S,

$$\forall (r, s) \in R \times S : \gamma(q_R(r), q_S(s)) \leq \gamma(q_R(R_{\max}), q_S(S_{\max}))$$

where $R_{\max}$ and $S_{\max}$ are the highest ranked items of their respective sets. If it can be asserted that all items in a join result must have a score greater than $s_{\min}$, then knowing the maximum scores of R and S is enough to determine if any joined items of R and S can be part of the join result. If the results of the join cannot be part of the result, there would be no need to evaluate pairs of items for inclusion in the result by computing the score and comparing it with $s_{\min}$. More importantly, there would be no need to compute the join of R and S in the first place, which may be computationally expensive.

4.2 Top-k joins

A basic method to perform a ranked join is performing a join then sorting the results. The issue with this method is that it requires the join to be completely evaluated before any part of the output can be produced. Joins can be computationally expensive and can produce result sets that are unnecessarily large for a top-k join. A method that can produce the same ranked join outputs using only a partially evaluated join can be more efficient by avoiding the cost of the full join. Such a method requires efficient navigation of the join space and must also guarantee that the join results that have not been evaluated are not part of the ranked join results.

A common type of relational join is the *equijoin*. Given two relations R and S and key functions k_R and k_S, we would like to find pairs of items whose keys under key_R and key_S respectively are equal. More formally,

$$\{(r, s) \in R \times S \mid k_R(r) = k_S(s)\}.$$

Traditional methods of processing such joins are nested loops, hash tables and sorting, all depending on the size of the inputs and what sort of indexes are available. However, these methods are most suited for producing full join results instead of partial results, usually in orders that are not particularly useful for ranked joins. Thus, by performing a ranked join on top of a regular join, a lot of work would be wasted evaluating parts of the join that do not affect the result of the ranked join.

4.2.1 Pull/Bound Rank Join

Research into ranked joins focuses on efficiently navigating the *join space* to avoid doing unnecessary work to process the join. The Pull/Bound Rank Join framework [18] captures an important aspect of the partial evaluation of joins for ranked joins. A join can be evaluated incrementally in a desired order by incrementally reading inputs (pulling) and joining newly pulled items with previously pulled items. At each step, one item from one input must be chosen and joined with previously pulled items from other other input. A specific order for the join can be achieved by carefully selecting which item in which input to pull next.

For a rank join with a monotone aggregate function, reading each input in descending order of rank allows establishing an upper bound for the score of any join results that have not yet been found. The upper bound T, also called the threshold, is defined as

$$T = \max\{\gamma(R_{\min}, S_{\max}), \gamma(R_{\max}, S_{\min})\}$$

where $R_{\min}, R_{\max}, S_{\min}$ and $S_{\max}$ are the minimum and maximum scores of items that have been pulled from R and S so far.

The intuitive explanation is that the next item of R must have at most the same score as the previous item, and may join with the item of S with the highest score, creating an aggregate score of at most $\gamma(R_{\min}, S_{\max})$ and vice versa for the next item of S. At any point during processing, all known join results with a score above this threshold are guaranteed to be the first results of the join in order of descending rank, because any unknown join result must have a score below the threshold. The join terminates when all join results have been produced by exhausting the inputs, but being an incremental join, it may be terminated earlier when enough results have been produced for a top-k join.

Hash-based rank-join

Ilyas et al. [11] proposed hash-based rank-join (HRJN), which uses two ranked relational inputs to perform a ranked equijoin with a monotone aggregate function. Its key feature is that it incrementally produces outputs by evaluating the join in an order that optimizes for producing join results with high aggregate scores first. The algorithm reads input items in order of descending rank to incrementally evaluate the join and places join results in a heap ordered by aggregate score. It keeps track of the upper bound for the score of any joined tuple that has not yet been found. Pulling input items causes the upper bound to decrease. When the item on top of the heap has a score greater than the upper bound, it can be removed and output as the next item in order of descending rank. A top-k query can be performed by using the k first outputs, which may not require evaluating the full join.

HRJN performs an equijoin, meaning it joins entries by equality of a key, so that the join condition is $\phi(r, s) : key_R(r) = key_S(s)$ where key_R and key_S get the keys of each input item. The join is performed using a hash table per input to index pulled items by their keys. Newly pulled items are placed into their respective hash tables and joined with indexed items from the other input by probing the other hash table. The algorithm can be made more generic with the realization that the use of hash tables as indexes is only a particular choice for the key equality join predicate. The hash tables can be replaced by other indexes such as spatial indexes for other kinds of join predicates. HRJN is considered an instantiation of the Pull/Bound Rank Join framework and can be further generalized as part of the Score-First Paradigm [16].

4.2.2 Parallelizing top-k joins

Pull/Bound Rank Joins such as HRJN are inherently sequential, and thus may not parallelize well. Parallelism could theoretically be achieved by having multiple threads pulling and processing input items in parallel. In order for a parallel join operation to be correct, adding an item to a set of previously pulled items and probing the other set of previously pulled items must be an atomic operation. Otherwise, some join results may be lost or duplicated. If an item is inserted into the set of previously pulled items before probing the other set, the following sequence of events may happen, causing an expected result (r, s) to be duplicated:

1. Thread T_1 pulls an item r from input R and inserts r into the set of previously pulled items of R.
2. Thread T_2 pulls an item s from input S and inserts s into the set of previously pulled items of S.
3. Thread T_1 probes the set of previously ready items of S and produces the tuple (r, s).
4. Thread T_2 probes the set of previously ready items of R and also produces the tuple (r, s).

Alternatively, if the other set is probed before the item is inserted, the following sequence of events may happen, causing an expected result (r, s) to be lost:

1. Thread T_1 pulls an item r from input R then probes the set of previously ready items of S, producing nothing.
2. Thread T_2 pulls an item s from input S then probes the set of previously ready items of R, producing nothing.
3. Thread T_1 inserts r into the set of previously pulled items of R.
4. Thread T_2 inserts s into the set of previously pulled items of S.

Because most of the work of the join is captured in this atomic operation, it may seem that the operation can barely be parallelized at all, but it is possible to move the probing out of the atomic operation. By redefining the atomic operation as one that inserts an item into a set of pulled items of one input and returns the set items to probe, the probing can be carried out separately. By capturing the set of pulled items from the input to be probed, the probing operation can ignore any items that are pulled concurrently during probing. By doing more work outside the atomic operation, it would be possible to achieve some degree of parallelism. This make a parallel implementation more suited for more expensive probing operations, but this is counter to the design of most applications of Pull/Bound Rank Joins which attempt to minify the cost of probing. Capturing the set of pulled items to probe in the atomic operation could also be computationally expensive. Instead of capturing a full set, the atomic operation could instead return a more lightweight handle on the current state of the set that can be used to ignore any items that are added concurrently.

Probing outside the critical section may require concurrent reads and writes to data structures, which may require synchronization mechanisms. In HRJN, the hash tables would require synchronization mechanisms for concurrent reads and writes, which would result in additional overhead. The collection of results will also require synchronization because multiple threads will be producing their own outputs. Using a more primitive indexing data structure for items that have been pulled (or no indexing data structure at all) could reduce the synchronization overhead, but the advantages of data structures such as hash tables would be lost.

Parallelism may alternatively be achieved by having multiple threads cooperatively pulling a bulk of item from an input, indexing them and cooperatively probing it against previously pulled items from the other input. Bulk operations are more likely to parallelize well than operations for individual items, but may require tight cooperation.

Divide-and-conquer

It is possible to perform top-k queries and top-k joins by using a divide-and-conquer strategy. The problem can be broken into subproblems by splitting the input into partitions and finding the top-k items of each partition. The top-k items of a set R can be found by dividing R into n non-overlapping partitions $R_1, R_2, \ldots, R_n$, finding the top-k items of each R_i, concatenating the top-k items of each partition, then finding the final top-k items among the concatenated results. By representing a top-k query as a function top_k that returns the top-k items of a set, it can be expressed as

$$top_k(R) = top_k(top_k(R_1) \cup top_k(R_2) \cup \cdots \cup top_k(R_n)).$$

A top-k join on R and S can be similarly processed using a divide-and-conquer strategy by considering it as a top-k query on $R \times S$. The total amount of items in $R \times S$ can be very large, so it is preferable avoid explicitly materializing $R \times S$ to partition it. One solution is to partition R into n partitions $R_1, R_2, \ldots, R_n$ and S into m partitions $S_1, S_2, \ldots, S_m$. Each pair of partitions R_i and S_j creates a subproblem $R_i \times S_j$. This creates nm total non-overlapping subproblems covering the entirety of $R \times S$.

Dividing the problem into subproblems and processing them in parallel is an opportunity to perform parallel top-k queries, but work efficiency may become an issue. If each subproblem can be processed in parallel in less time than the whole problem, it should produce a result in less time, but may result in wasted work. Divide-and-conquer leads to overall wasted work for top-k joins [22]. If the work required to process a two-way join without breaking into subproblems is w, then breaking it into n subproblems requires $\frac{1}{\sqrt{n}}w$ work per subproblem, for a total of $\sqrt{n}w$ work. In other words, the total amount of work increases by a factor of the square root of the amount of subproblems. Thus, the problem cannot be divided too much without risking a general loss in performance.

4.3 Ranked spatial queries

Spatial queries may be augmented to require rank-based filtering or sorting of results. A naïve method to achieve this would be a two-step process of performing the spatial query first, then filtering or sorting the spatial query results second. However, the R-tree range search algorithm can be modified to efficiently achieve both rank-based filtering and sorting in a single step. The modified algorithm operates on a MAX aR-tree, an augmented version of the R-tree, where each node contains the maximum score value of any object contained by the node. The MAX aggregate function has properties that allows both for filtering and sorting of query outputs using a modified range search algorithm.

4.3.1 aR-trees

The aR-tree is an R-tree that has been augmented with aggregate values assigned to each node. Similar to how the MBB of a node is an aggregation of the boxes of all objects placed under it, other types of object attributes can be aggregated to enhance the search capabilities of the tree for non-spatial attributes. An example aR-tree is the IR-tree [13] which is augmented for spatio-textual searches by storing aggregated inverted files in each node. A simpler example is an aR-tree that stores the maximum and minimum value of an attribute for all objects under the node, which may have practical applications in databases. For ranked spatial queries, we are mostly interested in aggregate values that can be used to determine bounds for score values.

The aggregated attribute of an aR-tree node $N.agg$ is determined by a decomposable aggregate function γ. For leaf nodes, the aggregate value is computed as the γ-aggregation of an attribute of each contained object. For inner nodes, the aggregate value is computed as the γ-aggregation of the $N.agg$ values of its children. Because γ is decomposable, it follows that the aggregate value of an inner node is the aggregation of all objects contained by the node, thus the aggregated attribute of any node is aggregation of all objects contained by the node.

The aggregate values stored in the aR-tree augment the search capabilities of the R-tree because each node carries information that can be used to make assertions about the properties of contained objects. By using $\gamma = \text{MAX}$ on an attribute a of each object, when searching for objects with the condition $O.a \geq v_{\min}$ it can be asserted that only nodes with $N.agg \geq v_{\min}$ can contain objects satisfying the condition. The MAX aggregate function also guarantees that the entry with the greatest attribute value can be found by descending the tree and always searching the node with the greatest aggregate value.

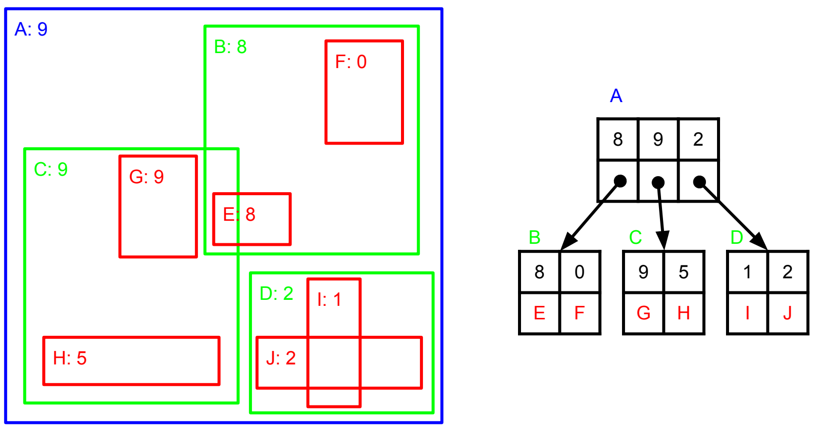

Figure 4.1: MAX aR-tree

4.3.2 Ranked range queries on aR-trees

The R-tree range search algorithm can be modified to only output objects with a score greater than a value $s_{\min}$ by using a MAX aR-tree and adding an additional condition for pruning subtrees. The modification is based on the principle that any aR-tree node with MAX score lower than $s_{\min}$ can be pruned from the search because the subtree cannot contain any objects satisfying the rank condition. This is based on the same principle that allows pruning subtrees from the search for not intersecting with the query box because the subtree cannot contain any object that intersect with the query box.

The R-tree range search algorithm can also be modified to output results in order of descending rank by dequeuing queue items in order of descending score. The score of an inner node entry is the MAX score of the node, while the score of a leaf node entry is the true score of the object. The principle is that in the queue of a range search, the highest ranked object that has not yet been found may either be contained by the inner node entry with the greatest MAX score, or is the leaf node entry with the greatest score. By always dequeuing the highest ranked queue entry, queue entries are processed in best-first order which guarantees that leaf entries will be visited in descending rank order.

The option of incrementally retrieving spatial query results in order of score with a score threshold using an aR-tree is invaluable for top-k spatial query processing. A top-k query on a single aR-tree can be processed by retrieving only the k first objects of a ranked range search without processing the whole queue. In more advanced cases, a score threshold $s_{\min}$ may be used to incrementally retrieve objects over the score threshold, where it is possible to incrementally raise the score threshold while processing the query.

4.4 Top-k spatial joins

A top-k spatial join combines the aspects of a spatial join and a top-k join. Given two sets of spatial objects, a spatial predicate is used to join the sets, and the k highest ranked joined tuples form the result. Top-k spatial join methods can be driven by the score and the space of the inputs. Score-driven methods for top-k joins may not be able to properly utilize the spatial properties of the inputs to perform the join. Similarly, space-driven methods for spatial joins may not be able to properly utilize the ranked properties of the inputs to rank the output. This difference calls for specialized methods that can utilize both.

A basic approach is to use a spatial join as a basis to join the inputs, then ranking the join result to produce the top-k results. This approach can fully utilize the spatial properties of the inputs, but suffers from the issues with doing unnecessary work to evaluate parts of the join that will not be part of the result. Methods such as the basic R-tree join and plane sweep-based joins [2] do not evaluate joins in useful orders for ranked joins. Therefore, the full spatial join has to be computed, which may use more memory and processing time than necessary to produce the top-k result. By applying the principles of Pull/Bound Rank Join by incrementally evaluating the join until it can be determined that any joined tuple that has not yet been produced cannot be part of the top-k result, it would be possible to avoid the wasted work. This requires an efficient method to evaluate partial spatial joins.

Qi et al. [17] proposed and evaluated several algorithms to perform top-k spatial distance joins on ranked inputs with monotone aggregate score functions. All algorithms are based on aR-trees using MAX as the aggregate function. The Score-First Algorithm (SFA) is an instantiation of the Pull/Bound Rank Join framework that utilizes aR-trees for indexing pulled input values. The Distance-First Algorithm (DFA) requires both inputs to be fully indexed by aR-trees and performs a ranked R-tree join. The Block-based Algorithm (BA) works similarly to the SFA by prioritizing objects with high scores first, but pulls inputs in blocks to bulk load aR-trees and probes in a series of aR-tree joins.

4.4.1 Score-First Algorithm

The Score-First Algorithm (SFA) is an instantiation of the Pull/Bound Rank Join framework by pulling input items in descending order of score and establishing a score bound. Being a spatial join algorithm, it uses dynamically constructed aR-trees to index pulled items via R-tree insertion, and probes pulled items with ranked range search. When the algorithm has found at least k candidate results, the score of the k^{th} candidate result is a threshold for the score of any results that have not yet been found that may replace the top-k candidates. SFA can use the threshold to prune subtrees when probing the aR-tree to only find join results that will become top-k candidates.

With SFA, pairs of objects with high scores are found fast, and the results are expected to be found fast if the objects with the highest scores are close to each other, but it is generally not as efficient as other alternatives. With little correlation between scores and locations, many entries have to be dynamically inserted and many range queries have to be performed if the objects of the highest ranked pairs reside deep in the inputs. Compared to bulk loading methods, the dynamic insertion used in SFA may be expensive or produce R-trees with bad query performance. The overhead of maintaining and probing the R-trees also increases with each pulled item as the cost of each insertion and range query increases as the R-trees grow larger.

4.4.2 Ranked spatial join on MAX aR-trees

As part of DFA, Qi et al. [17] described an algorithm can be used to perform a ranked spatial join on ranked inputs indexed by MAX aR-trees. It uses aggregate scores on each aR-tree node to determine upper bounds for the scores of pairs of objects. If the score of a pair of objects (r, s) is defined as $q(r, s) = \gamma(q_R(r), q_S(s))$ where γ is a monotone aggregate function and $q_R(r)$ and $q_S(s)$ are the scores of the objects r and s, then an upper bound for the score of all pairs of objects that can be produced by joining two nodes can be determined. By considering the objects contained by each node as sets, an upper bound for the score of all pairs of objects that can be produced by joining nodes R and S is

$$\gamma(\max\left(q_R(r) \mid r \in R\right), \max\left(q_S(s) \mid s \in S\right))$$

The MAX score values for the set of objects contained by a node is included as an attribute on each aR-tree node, thus it is trivial to compute the upper bound for any pair of nodes.

The ranked spatial join is a variant of the R-tree join algorithm. Due to the similarities between the range search algorithm and the R-tree join algorithm, both algorithms can be similarly augmented to include a score threshold and can also order its output by score. Recalling that the R-tree spatial join algorithm operates on a queue containing pairs of node entries, the ranked variant assigns a score to each pair of node entries. The score of a pair of inner node entries is the γ-aggregate of the MAX score of each node, while the score of a pair of leaf node entries is the true score of the pair of objects under γ. This score can be used to prune or sort pairs of node entries.

The R-tree spatial join algorithm can be augmented with a score threshold $s_{\min}$ so that it only outputs pairs of objects with a score greater than $s_{\min}$. It follows the principle that any pair of node entries whose score is below $s_{\min}$ is either a pair of inner node entries that cannot produce pairs of objects with scores above the score threshold, or is a pair of leaf node entries whose objects cannot be a pair with score above the score threshold. Thus, any pair of node entries with score below $s_{\min}$ can be pruned from the join. This additional condition is similar to how pairs of node entries can be pruned from the search for not intersecting with each other. This method efficiently avoids doing unnecessary work to join objects based on both their scores and spatial attributes.

The R-tree spatial join algorithm can be modified to output pairs of objects in descending order of score by always dequeuing the pair of node entries greatest score. It follows the principle that in the queue of a spatial join, the pair of objects with the greatest score that has not yet been found may either be produced by the pair of inner node entries with the greatest score, or is the pair of leaf node entries with the greatest score. By always dequeuing the pair of node entries with the highest score, queue entries are processed in best-first order which guarantees that pairs of leaf node entries will be visited in descending order of score.

4.4.3 Distance-First Algorithm

The Distance-First Algorithm (DFA) is performed as a ranked spatial join on MAX aR-trees that terminates when the first k results are found. Each input must be fully indexed by a MAX aR-tree, either using a previously constructed index or by bulk loading an aR-tree prior to the join. When evaluating DFA against other algorithms, the cost of constructing the aR-trees should be accounted for, as bulk loading aR-trees prior to the join may be an expensive operation.

DFA works well if there is a correlation between the scores of the objects and their locations. Because the aR-tree entries are clustered by their locations and not their scores, pairs are pruned primarily by the spatial predicate, and not their ranking. The objects with the highest scores that are close to each other are identified fast, while the rest of the results may require expanding many pairs of nodes before termination.

4.4.4 Block-based Algorithm

The Block-based Algorithm (BA) is similar to SFA by prioritizing objects with high scores, but instead of pulling one object at a time, it pulls inputs in blocks. Each pulled block is bulk loaded into an aR-tree then joined with all the blocks that have been pulled so far from the other input using a series of ranked aR-tree joins. BA keeps track of candidate top-k results, and can also use the score of the k^{th} candidate result to prune subtrees when performing each spatial join.

BA is an attempt to overcome SFA's failure to quickly find spatial join results, and DFA's failure to quickly find pairs with high scores. Like the SFA, the cost of probing in the Block-based Algorithm increases with each pulled item, although it increases linearly with each block instead of logarithmically with each item. Still, the BA generally exhibits better performance than the SFA due to a more efficient joining method designed to terminate before the cost increases too much. When considering the computational cost per item in the inputs, performing joins at block level is more efficient than performing joins one item at a time because the spatial characteristics of both inputs can be used more efficiently. Indexing blocks with bulk loading is more efficient than dynamic insertion of each item, and joining aR-trees is more efficient than probing once per pulled item. The number of aR-tree joins that have to be performed per loaded block can be limited by observing the upper bound for the score of any results that can be produced by joining a pair of aR-trees to avoid any unnecessary joins.

4.4.5 Parallelizing top-k spatial joins

The Score-First Algorithm is not a target for parallelization due to its sequential nature, its use of mutable R-trees and the complexity and inefficiency of supporting concurrent aR-tree insertions and range searches. The performance of concurrent R-trees falls quickly with the amount of insertions per range search [10]. Most GPGPU research on R-trees is based on read-only R-trees, which makes the development of mutable R-trees a challenge.

Ranked aR-tree joins may be parallelized like other branch-and-bound algorithms. It is based on fine-grained priority queue operations, potentially with a significant amount of enqueued items per dequeued item depending on the fanout and the spatial predicate. A number of concurrent priority queues have been devised for branch-and-bound algorithms that could be applied to the aR-tree ranked join to allow multiple threads to enqueue and dequeue items from a shared priority queue.

Priority queues for GPGPU are much more constrained, however, and have received limited research.

The Distance-First Algorithm is a more likely candidate for parallelization. The initial bulk loading phase can be parallelized by bulk loading both inputs in parallel, also using parallel bulk loading algorithms. Bulk loading algorithms such as STR are based on sorting, for which a number of parallel algorithms exist, both for CPU and GPGPU. The aR-tree join can be parallelized as described earlier.

The Block-based Algorithm is uniquely parallelizable by consisting of a series of smaller, mostly independent tasks. First of all, the initial sorting of the inputs can be parallelized. The rest of the algorithm can be parallelized using multiple methods. One method involves following the sequential steps, but parallelizing each step with parallel bulk loading and parallel ranked aR-tree joins. Another method involves bulk loading and joining multiple aR-trees in parallel, only synchronizing to pull blocks and to gather results.

Chapter 5

Sorting on CPU/GPU

Sorting is a vital part of the implementation of top-k spatial joins, both for sorting items in order of score and for the sorting used to bulk load R-trees in the Sort-Tile-Recursive algorithm. This chapter covers some sorting algorithms for CPU and GPU and the family of heap data structures used to keep items in order. Bulk loading based on the Sort-Tile-Recursive algorithm requires a series of sorting passes. The Block-based Algorithm relies on a sorting algorithm for prior sorting of inputs. A priority queue used for ranked spatial joins may be based on heaps, and top-k results may also be accumulated in min-heaps.

Which sorting algorithm should be used depends on the desired parallelism, the input size and whether the sorting will be carried out on GPU or on CPU. Quicksort and its sibling introsort are particularly popular in CPU environments, and its design allows for some optimizations that could be relevant to the Block-based Algorithm. Radix sort is popular in GPU environments and is easily programmable and available in CUB, a template library for CUDA. A particularly interesting option on GPUs is the use of sorting networks.

5.1 Sorting algorithms

Sorting is an essential part of many algorithms, and thus is the subject of a large volume of research. This section covers some prominent sorting algorithms for single-threaded and multi-threaded systems as well as sorting algorithms for GPU. Indeed, a number of sorting algorithms have been adapted for GPU and can outperform sorting on the CPU. The ranking and applicability of various GPU sorting algorithms can depend on a number of things [19]. Which sorting algorithm is optimal depends the amount of arrays to sort, their sizes, and which resources that can be allocated to sort them. For instance, small arrays that can fit in shared memory in a CUDA thread group may be optimally sorted using a different algorithm than arrays residing in global memory.

5.1.1 Quicksort

Quicksort is a divide-and-conquer sorting algorithm. Given an array of unsorted items, the main procedure consists of selecting an item called the *pivot* then partitioning the array into one sub-array of items ranking below the pivot and another sub-array of items ranking above the pivot. These sub-arrays are then sorted recursively using the same procedure. Some variants of quicksort may also select multiple pivots instead of just one to divide the array into more, smaller sub-arrays. Quicksort is often used indirectly via introsort, which is a popular variant of quicksort that delegates to alternative sorting algorithms for sub-arrays when the recursion goes beyond a certain threshold or when the amount of items below a threshold to improve general performance and the worst-case performance.

The choice of pivot is important for the performance of quicksort, because it operates optimally when the pivot is exactly the median of the sub-array. The median of medians is often used as an approximation, which is a method that samples groups of items from the array, calculates medians for each group, then calculates the median of the medians of each group.

Quicksort can be implemented both with and without parallelism. Task parallelism can be achieved by considering each operation to partition an array into sub-arrays as a task that produces two more tasks to sort each sub-array. Each task is associated with an independent part of the array, thus each task can be processed in parallel without conflicting reads and writes to the same positions in the array. Data parallelism can also be achieved for each task by comparing data items with the pivot in parallel. The placement of items in the resulting partitioned sub-arrays is dependent on the value of all items in the sub-array, thus some communication may be necessary to compute the placement of each item in the resulting sub-arrays. The placement can for instance be computed with a parallel prefix sum with an external array.

In a parallel CPU environment, the potential for task parallelism is generally dominant for quicksort except for in the earliest phases when the amount of tasks is lower than the amount of threads. Assuming pivots are selected fairly, each task produces two subtasks so that the amount of available tasks grows exponentially, quickly producing enough tasks for each thread to have its own task to work on in parallel. Compared to the synchronization required by data parallelism, the overhead of managing and distributing tasks is low, especially when each thread has its own task and can produce more tasks to process on its own.

5.1.2 Radix sort

Radix sort is a sorting algorithm that is particularly well suited for medium-to-large sized arrays in global memory on GPUs. What distinguishes radix sort from many other sorting algorithms is that it operates in $O(nw)$ time where n is the amount of items to sort and w is the amount of digits in each key. To apply this sorting algorithm, it must be possible to sort it by some key that can be divided into digits.

CUB supports sorting values by key in radix sort, with arbitrary value types and support for most primitive data types for keys via templating, and is therefore a useful library for applications. For integral key data types, cheap bitwise operations are used to group bits into digits for a configurable base which must be a power of two. The d^{th} digit in base b^2 can be found by right shifting the integer by db and using the first b^2 bits as the digit. Interestingly, CUB even supports floating point data types for keys, which would normally result in far too many digits for an efficient radix sort, but by exploiting the IEEE 754 binary representations with some bitwise transformations, floating point keys can be sorted like integral data types.

5.1.3 Bitonic merge sort

Bitonic merge sort is a parallel sorting algorithm that has $O(n \log^2(n))$ comparators but a delay of $O(\log^2(n))$ when comparators are applied in parallel. A comparator is a directed connection between two array positions l and r. When applied, the comparator ensures that the item in position l is smaller than the item in position r by swapping the items if it has to. A sorting network is applied by applying a series of fixed comparators, many of which can be applied in parallel. Due to the massively parallel nature of GPUs, it is a popular method for sorting small arrays. It is particularly applicable in shared memory.

5.1.4 Potential optimizations

To optimize the execution the Block-based Algorithm in general, some potential optimizations to the application of sorting have been identified. Some of these optimization options are used in the implementations.

Segmented sorting

A segmented sort operates on a single array divided into segments, and sorts each segment. Segmented sorting occurs naturally in the Sort-Tile-Recursive algorithm. In each sorting pass, STR partitions a number of large *slabs* into smaller slabs by sorting each large slab. Instead of considering this as one sorting operation per slab, it can be considered as one segmented sorting operation. The distinction between multiple sorts and a segmented sort can be applied for some optimizations. Segmented sorting may avoid the overhead of allocating and freeing resources for multiple invocations of a sort. It also implicitly enables task parallelism by considering the sorting of each segment as a set of tasks.

Block sorting

It is possible to optimize Sort-Tile-Recursive and the Block-based Algorithm using a relaxed sorting algorithm that arranges items into blocks. Most of the work performed by the Sort-Tile-Recursive algorithm lies in sorting sequences of R-tree records to partition them into fixed-size, spatially separated blocks. By fully sorting the sequence, the records in each block also become internally sorted. However, Sort-Tile-Recursive does not demand that the records inside each block are internally sorted. The records of each block will subsequently either be assembled into nodes or be sorted again by a different key, where the prior order of the records does not matter. The Block-based Algorithm similarly pulls blocks from sorted inputs, where the internal order of items in each block does not matter, because they will immediately be bulk loaded into aR-trees.

We define the block sorting problem. Given a sequence S and a block size r, we would like to partition S into a sequence of $n = \frac{|S|}{r}$ r-sized blocks $B_1, B_2, \ldots, B_n$, where each block B_i contains the $((i-1)r+1)^{th}$ to $(ir)^{th}$ ranked items of S. An item is the t^{th} ranked item if it is the t^{th} item of the sequence produced by fully sorting S.

A trivial but work-inefficient method to perform block sorting is by performing a regular sort then dividing the array into blocks of size r. This naturally arranges the items into their exact positions by rank, but the work spent on internally sorting the items in each block is wasted and could be avoided with a relaxed sorting algorithm. The amount of work that can potentially be saved per input item should be proportional to the block size.

A related class of algorithms are partition-based selection algorithms, which are algorithms for finding the k^{th} ranked item in an array by partitioning the array around the k^{th} ranked item. The k^{th} ranked item is placed as the k^{th} entry in the array, dividing the array into two sub-arrays, one on each side of the item. Items placed in the sub-array before the k^{th} item are items ranking before the item, and items placed in the sub-array after the k^{th} item are items ranking after the item. Partition-based selection algorithms and block sorting algorithms are similarly related to sorting algorithms with the goal of avoiding work by relaxing the internal order of each partition. Divide-and-conquer partition-based selection algorithms like quickselect and introselect work similarly to their sorting algorithm counterparts, quicksort and introsort, but will only process a subtask if its range of items contains the k^{th} item. The order of items in partitions not containing the item is unrestricted and can thus be ignored.

A particularly relevant variant of the partition-based selection problem is the partition-based multi-selection problem, where multiple items are being selected simultaneously. A block sorting problem can be modeled as a partition-based multi-selection problem of selecting every r^{th} item where r is the block size. A divide-and-conquer partition-based selection algorithm can be modified to only process a subtask if its range of items contains any of the items being selected.

Incremental sorting

Normally, the inputs of the Block-based Algorithm would be fully sorted prior to prior to running the algorithm, which constitutes a fixed up front processing cost which minimizes the processing cost of pulling a block. Knowing that the algorithm strives to terminate as early as possible by pulling as few blocks as possible, it could be more efficient to offset the processing cost to the operation of pulling a blocks.

An alternative method to retrieve items in descending order of score is by iteratively retrieving items from a max-heap. The heap is constructed prior in $O(n)$ time, then each item can be retrieved in $O(\log n)$ time. This offsets some of the cost of sorting from prior processing to the process of retrieving each item. The total amount of work used to retrieve items in descending order of score is then proportional to the amount of items that are retrieved. Retrieving every item would have a similar cost to a heapsort.

The incremental sorting problem is a variant of the partial sorting problem. Partial sorting retrieves the k smallest or largest items of an array in order, which can for instance be used for top-k queries. For an incremental sort, we are interested in incrementally increasing the amount of items that are retrieved by a partial sort.

Quicksort can be modified to become an efficient incremental sorting algorithm by storing information about the partitioning of items and incrementally working on the first partition of unsorted items until the desired amount of items have been sorted [15]. A naïve implementation of incremental sorting based on quicksort would be to perform a partial quicksort for the next m items each time k is increased by

m. It will be observed that each time a partial quicksort is performed, a number of items beyond the ones returned from the partial sort will be moved around and placed into partitions. The information about these partitions would be lost each time the partial sort is complete, therefore work is wasted on re-partitioning already partitioned items each time another partial sort is performed. By storing information about the partitioning, each partial sort can pick up where the last one left off.

It might be possible to parallelize incremental quicksort similarly to how a regular quicksort is parallelized, but not without tradeoffs. The amount of parallelism for the minimal amount of work required to sort another m items from an array of n items is mostly limited by m, not n. Task parallelism would be achieved only for sub-arrays smaller than m, and as the sub-array size trends towards m, the level of data parallelism decreases quickly. Another alternative would be an asynchronous incremental sort where one thread incrementally sorts the array and reports its progress to consumer threads.

5.2 Heaps

A heap is a balanced tree structure that satisfies the *heap property*: The value of a parent node is smaller than or equal to the value of its children. More generally, a heap satisfies the heap property with respect to a desired order. The min-heap follows an ascending order, matching the previous description of the heap. The max-heap follows a descending order, so that the value of a parent node is larger than or equal to the value of its children. The heap property ensures that the smallest or largest value in the heap always resides in the root node. Thus, using a heap to contain a set of ranked values makes it trivial to find the smallest or largest value by looking at the root node. The operations that modify the heap must only maintain the heap property, which is generally cheaper than maintaining a fully sorted list.

The basic operations that a heap is designed for is *pop* and *insert*, which removes and inserts values in the heap respectively. Depending on what type of heap it is, pop retrieves and removes either the smallest or the largest value in the heap. Insert places a new value into the heap. We can also consider the *peek* and *pop-insert* operations. Peek simply returns the smallest or largest value in the heap without removing it. Pop-insert has the same effect as a pop followed by an insert, which effectively replaces the smallest or largest value in a single operation. It differs from performing a pop and an insert in sequence by allowing optimized implementations.

Heaps are the basis for creating priority queues, which are essential for branch-and-bound algorithms. If a heap contains a set of tasks with priorities, a max-heap can order these tasks by descending priority, so that the task with the greatest priority in an ever changing set of tasks can always be retrieved. If processing a task produces additional tasks, they can be inserted back into the heap.

5.2.1 Parallel heap

The parallel heap [7] is a heap variant where each node contains multiple values and where each node can have many children. Instead of satisfying the usual heap property, it satisfies a *heap-like property*: The *smallest* value of a parent node is smaller than or equal to the *smallest* value of its children. The parallel heap supports the same operations that a regular heap does, but is designed specifically for CUDA.

Even though it is based on device memory, it creates efficient device memory access patterns and uses the parallel execution and efficient cooperation within warps to its advantage.

The parallel heap is designed so that a warp of threads cooperatively owns the heap, where all owning threads must cooperatively participate in each insert and pop operation. This distributes the work of each operation between the threads, but also means that operations are not concurrent.

The implementation of the parallel heap revolves around an efficient method of inserting an item into a sorted array of 32 values using warp ballots and warp shuffles. Each thread in the warp uses private memory to hold its own value from the array, so that the first thread holds the first value and the last thread holds the last value. The method places the value into the correct position in the array by shifting the following values along and popping the last value as an excess value. First, each thread compares the item to be inserted with its own value in parallel. The comparisons are used as predicates for a ballot, resulting in a 32-bit integer whose N^{th} bit is set if the N^{th} value in the array is smaller than the value to be inserted. This results in an integer whose binary representation may be 00000001111111111111111111111111, which indicates that the value is greater than the first 25 values in the array, and smaller than the remaining values. The value therefore should be placed in the 25^{th} position, after all smaller values and before all larger values. Generally, the value will be inserted into the position following the most significant 1. The *bfind* PTX instruction can be used to retrieve the position of the most significant 1 to determine where the value should be inserted. A shuffle up is then used to shift the following values between threads, and a shuffle may also be used to broadcast the excess value to all threads if it should be used in later processing.

A particularly efficient choice for the implementation of multiple values per node is therefore using an arrays where each thread stores a value from each node. Each node in the parallel heap has a sorted array of 32 values, where the first value (belonging to thread 0) is the smallest value. Thus, to satisfy the heap-like property, the first value in the array of a parent node must be smaller than or equal to the first value of the arrays of its children. For a parallel heap with capacity for n nodes, the values are laid out in memory so that each thread has its own local array of n values to store a value from each node.

Each thread uses local memory to store its own value from each node array, while a copy of the smallest value of each node is stored in shared memory to accelerate heap operations. Since local memory is stored in device memory, the capacity of the heap is not limited by the availability of registers or shared memory in each SM, but it suffers from the latency of accessing device memory. Given that the values are stored in local memory, threads can only access their own values, and must otherwise use shuffles to access values from other threads. By storing the first value in each array, thread 0 is given the special responsibility of storing the smallest value in each node, which is often used for heap operations. Thread 0 must therefore maintain the smallest values in shared memory, so that all threads may cooperate for heap operations involving the smallest value of each node without requiring shuffles and device memory accesses for each value.

The parallel heap has an unusually large arity of 32, meaning that each parent node has 32 children. This significantly decreases the height of the heap but increases the work required to determine which child of a parent node has the smallest value.

Decreasing the height of the heap results in fewer hops between the bottom and the top of the heap during heap operations, which decreases the cost of accessing the local memory when hopping between nodes. When the heap has 1056 items or less, only one hop is required between the root node and each leaf node. The child with the smallest value can be found using parallel warp-level reduction on the smallest values, all of which can be retrieved from shared memory. This makes retrieving the child with the smallest value comparatively efficient.

5.2.2 Top-k accumulator min-heaps

Heaps can be used to accumulate the top-k items for a sequence of items. Given a sequence of ranked items and a heap with capacity for at most k items, the set of top-k items for items pulled from the sequence is recorded incrementally in the heap with each pulled item. The first k pulled items are inserted into the heap unconditionally until the heap is at capacity. When a new item is pulled when the heap is at capacity, it must replace the minimum item in the in the current set of top-k items with a pop-insert, but only if it is greater than the current minimum item in the heap. Using a min-heap, the minimum item always resides at the top, therefore any new item has to be compared with an item that is found in constant time. If the new item is greater than the minimum item in the heap, the former minimum item in the top-k result is replaced in $O(\log k)$ time.

The cost of accumulating the top-k result depends on how often items have to be replaced. In the worst case, the sequence contains items in increasing order, which requires replacing the minimum item each time a new item is pulled. In the best case, the sequence contains items in descending order, which never requires replacing the minimum item. Generally, the more the sequence trends towards a descending order, the better the accumulation performs as fewer swaps have to be made to maintain the heap property.

A top-k accumulator min-heap not only incrementally determines the top-k items, it also incrementally determines an increasing lower bound for the full top-k results. When at least k items have been pulled, the minimum item in the heap is a lower bound for any items not pulled so far that may become part of the top-k result. When the items of the sequence are produced incrementally, knowledge about the lower bound can be used to prune sets of items that cannot be part of top-k result. If it can be determined that there are no more items to pull above the lower bound, the heap contains the full top-k result and the accumulation may terminate.

The current top-k items can be retrieved in $O(k \log k)$ time by iteratively popping each item from the heap, similar to the second phase of heapsort. For an array-based heap, the top-k items can be retrieved in-place in descending order by repeatedly swapping the last item in the heap array with the first, shrinking the heap by one then propagating down until the heap is empty.

Chapter 6

Implementation

This chapter describes a single-threaded, a multi-threaded and a CUDA implementation of the Block-based Algorithm for top-k spatial joins based on the concepts and methods described in previous chapters.

The CUDA implementation of the Block-based Algorithm is based on the work in the project preceding this thesis [1]. The project laid the groundwork for a CUDA implementation of the Block-based Algorithm, with some parts that were implemented and some parts that were planned, but did not deliver anything measurable. This chapter describes a complete CUDA implementation of the Block-based Algorithm in more detail.

The single-threaded and multi-threaded implementations were developed to compare with the CUDA implementation. The single-threaded implementation is mostly faithful to the original description of the Block-based Algorithm by Qi et al. with some implementation details that are shared between all implementations. The multi-threaded implementation is an adaptation of the Block-based Algorithm that parallelizes both the prior sorting and the joining of blocks. The multi-threaded implementation and the CUDA implementations parallelize in similar ways, but the CUDA implementation does it to a much higher degree.

We consider each input as an array of items, each with an MBB, a score value and a data value, and the output as an ordered array of up to k items, each with a score and a pair of data values. An item in the output array is created by joining a pair of input items, computing and storing their aggregate score according to the score aggregate function γ and storing the data value of both input items. Only pairs of input items whose MBBs satisfy the spatial join predicate ϕ may be joined to produce an output item.

The implementation is written in C++ with the extensions to the language provided by CUDA. Single-threaded and multi-threaded implementations are written following the C++ 17 standard, using a few modern features that cannot be used with CUDA. The CUDA implementation is written following the C++ 14 standard, which is the most modern version of C++ supported by CUDA at the time of development.

Templating is used to support various dimensionalities and various combinations of data types for input and output data. The *element type* is a numeric data type used to represent the bounds of MBBs such as single-precision floating point numbers or integers. The *key type* is used both for the score values of input items and the score values of output items, which is restricted to a set of integer and floating point data types. The *data type* is arbitrary, but should generally be a small data type to minimize the memory requirements. Therefore, if our top-k spatial join implementation should be used in an application that has a significant amount of data for each input item, the application should minimize the amount of data used in the top-k spatial join by using a simple data type such as integer keys to retrieve additional data from a database based on the data returned from the top-k spatial join.

For the sake of simplicity, this chapter describes an implementation of top-k spatial joins where higher scores are always better for both input items and output items for simple aggregate functions such as SUM, MAX or weighted sums. Using additional templating, the implementation actually supports different orders, both for each input and the output to support more complex score aggregate functions.

6.1 Linearized aR-tree

The aR-tree implementation is designed for device memory on the GPU, but is just as suitable for CPU usage. For top-k spatial joins, we will be executing ranked spatial joins on aR-trees. aR-trees will be traversed in order of score, causing scattered random reads to access the nodes in memory, which means operations on aR-trees on the GPU are expected to be memory bound. Therefore, special care is required to achieve good random read performance when reading aR-tree data from memory. Specifically, the reads should be aligned with the cache lines and memory transactions of the GPU.

Each item of a node in the linearized aR-tree, whether it is an inner node or a leaf node, is represented as a *record* in an array that contains all records of an aR-tree. The array is divided into h segments, each larger than the former, corresponding to the h levels of the R-tree. The segments can be laid out in any order. In the CPU implementation, segments are laid out in descending order of level so that the first segment of the array contains records belonging to the root node, and the last segment contains all the leaf node records of the R-tree. In the GPU implementation, the segments are laid out in ascending order of level. Each segment is divided into r-sized blocks, each block corresponding to one node at the segment's level in the aR-tree. The order of the blocks within each segment is unrestricted. By storing the entire R-tree in a single array, blocks can be addressed by their position in the array, which uses less memory than 64-bit pointers. It also simplifies the allocation of memory for the R-tree.

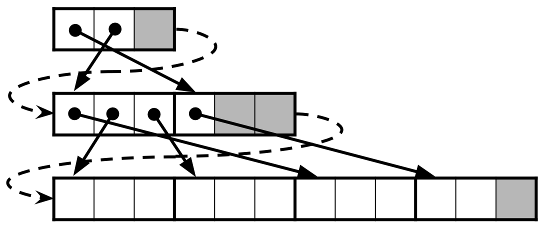

Figure 6.1: R-tree memory layout with $r = 3$

Each record in the array is either an inner node struct $\{mbb, score, idx\}$ or a leaf node struct $\{mbb, score, data\}$. Thus, all records have $\{mbb, score\}$ in common, while a union is used to store either the *data* or *idx* field. For leaf node records, the *mbb* and *score* fields contain the direct MBB and score of an entry in the R-tree, while the remaining *data* field is used to produce query results. The *data* field can, for instance, be an identifier for an item in a database that can be used to retrieve more data for query results. For inner node records, the *mbb* field contains the MBB of the node that it points to, and the *score* field contains an aggregation of the *score* of its children. The *idx* field of an inner node record identifies the block in the R-tree array that contains all the records of the node.

Each block has a fixed size of r records but is not necessary fully packed. A fully packed node has a block full of records, while a non-full node leaves its rightmost records uninitialized as padding. The bulk loading method may ensure that all blocks but the final block of each segment are fully packed, or that all blocks are fully packed. This ensures that the number of records in each node does not have to be stored. Instead, the total amount of records in a segment fully describes the distribution of records in blocks. The amount of blocks in a segment of size $|S|$ is $\left\lceil \frac{|S|}{r} \right\rceil$, where the amount of records in the final block is $((|S| - 1) \bmod r) + 1$. To access the MBBs and score (and child node references) of all the records of a node, all that has to be accessed is one fixed size block. By padding the nodes to a fixed block size, accessing all records of a node can also be done safely by accessing a whole block, which will never be out of bounds for non-full nodes. For a sufficiently large input, the size of the padding should be negligible.

The R-tree memory layout is designed for storage in device memory on the GPU by special alignment. The R-trees are in generally expected to be too large to reasonably fit in the limited space of shared memory and registers, especially when competing with other data structures for space. Therefore, special attention is given to the alignment and size of the blocks of the record array to optimize accessing the R-tree in device memory. If full blocks are accessed in bulk, the blocks should be laid out so that they are naturally aligned with the memory transactions between the L1 cache, L2 cache and device memory. This ensures that the access latency remains low. Therefore, a strategic value for the fanout r has to be chosen so that the size of each record multiplied by r is a multiple of the transaction size. Depending on the size of each record, this may also involve adding padding to each record.

The CPU application of linearized aR-trees is quite flexible due to the versatility of the CPU and its memory, unlike the CUDA application of them which significantly restricts the fanout and the amount of objects in an R-tree, and also requires both the fanout and the amount of objects in an R-tree to powers of two. The limitations on the CUDA application are imposed by the limited availability of shared memory and functions that operate most efficiently on inputs whose sizes are powers of two. More complex and potentially less efficient methods would have to be used to relax these restrictions, which could become future work.

6.1.1 Computing R-tree layouts

Prior to building an R-tree, a segment layout may be computed as a function of the size of the input n and the fanout of the R-tree r. Given an input size n and a fanout of r, the Layout Algorithm produces a segmentation of an R-tree array that fits n objects, expressed as a list of segment sizes and offsets. The layout is only a function of the size of the input and the fanout of an R-tree that is to be bulk loaded. The layout is not dependent on the actual data in the R-tree. Therefore, for repeated use of the same R-tree configuration on different R-trees, the segmentation can be precomputed and re-used.

Algorithm 6.1 Layout Algorithm. n is the amount of entries in the R-tree, and r is the fanout of the R-tree.

```
function LAYOUT($n, r$)
    Initialize L as list of $\lceil log_r(n) \rceil$ items
    $i \leftarrow 0$
    repeat
        $n \leftarrow \lceil \frac{n}{r} \rceil$
        $L_i$.size $\leftarrow n \times r$
        $i \leftarrow i + 1$
    until $n \leq 1$
    $o \leftarrow 0$
    repeat
        $i \leftarrow i - 1$
        $L_i$.offset $\leftarrow o$
        $o \leftarrow o + S_i$.size
    until $i = 1$
    return $L$
end function
```

The intuitive explanation for the Layout Algorithm is that n R-tree node records must be distributed into at least $\lceil \frac{n}{r} \rceil$ nodes if the fanout of the R-tree is r. For an R-tree with n objects, n leaf node records exist on the bottom level. The leaf node records are distributed into $\lceil \frac{n}{r} \rceil$ leaf nodes, meaning there must be $\lceil \frac{n}{r} \rceil$ inner node records on the second level. The inner node records on the second level must be similarly distributed into $\left\lceil \frac{\lceil \frac{n}{r} \rceil}{r} \right\rceil$ inner nodes. If the amount of records in a level is below r, it must contain the records of the root node. Thus, the amount of records is divided by the fanout until the records form the records of the root node. In the basic case of $n \leq r$, the R-tree will only be a root, and the algorithm returns the layout of a single segment for the root with a size of r and an offset of zero. When $n > r$, the algorithm returns a layout of multiple segments. The algorithm computes the offset of each segment as the cumulative size of previous segments. Because L_1 is the last segment of the array, the total size of the array can be computed as L_1.offset + L_1.size.

The order of segments in the R-tree record array is of little technical importance, but the Layout Algorithm lays them out so that we intuitively "descend" into the array when descending down the R-tree. The first segment of the array contains the root node records and the last segment of the array contains the leaf node records.

The Layout Algorithm is inherently sequential and is also not a target for parallelization. The computed layout is a small list that only has to be computed once for each R-tree configuration, thus the performance of the algorithm is largely unimportant.

6.2 Block sorting

Block sorting is an optimization for the sorting used in Sort-Tile-Recursive and the prior sorting in the Block-based Algorithm. The Sort-Tile-Recursive implementation uses segmented block sorting, while the Block-based Algorithm implementation does not. We describe a segmented implementation of block sorting, considering a non-segmented sort to operate on a single segment.

The CPU implementations of segmented block sort algorithms are based on the MSVC implementation of introsort. The algorithm uses the term *work item* for an object that identifies a segment of the array to partition which may, when processed, produce additional work items. When processing a work item, a pivot is selected using the median of medians approach and the segment is partitioned around the pivot. A work item may be created for each partition, but only if the partition overlaps multiple blocks. If the size of a work item is below a threshold it is simply sorted using insertion sort.

Single-threaded implementation

The single-threaded variant of the segmented block sort algorithm iterates over each segment and uses recursion to process work items produced by other work items. This keeps the collection of work items on the stack, which may overflow. The risk of stack overflow is low because the height of the stack is generally $O(\log n)$, but it may overflow in a worst-case scenario causing the recursion to go too deep. This is not accounted for in the implementation like it would be for introsort, but it should be viable to fall back to heapsort if there is a risk of encountering a worst-case input.

Multi-threaded implementation

The multi-threaded variant of the segmented block sort algorithm uses the same concept of work items from the single-threaded implementation, but is parallelized using a work stealing scheduling strategy. It starts with one shared work queue containing a work item for each segment and a (mostly) lock-free single-producer, multiple consumer work queue per thread. Each thread owns its own work queue and is the only thread that can produce items for that queue, but any thread may retrieve work items from the queue of any thread. A thread will primarily append and retrieve work items from its own work queue, but may also attempt to *steal* work items from any other work queue. Whenever a thread's work queue is empty, the thread will attempt to take a work item from the shared work queue to start processing the next segment, and if there is no work item to be taken, the thread will attempt to steal work from other threads. Work stealing is therefore only done when a thread cannot produce more work on its own. This offloads the work from congested threads, ensuring steady progress on the sort.

CUDA implementation

We have not implemented a CUDA version of block sorting. The algorithms relying on block sorts are instead based on normal sorting algorithms in CUDA. For small arrays such as the ones used in tile sorting, bitonic merge sort is used, operating entirely within shared memory. For large arrays such as the ones used to sort the inputs of the Block-based Algorithm, the radix sort from CUB is used.

6.3 aR-tree bulk loading

A major part of the computation in the Block-based Algorithm is a number of small bulk loads of aR-trees. Bulk loading can be performed both on the GPU and on the CPU with slightly different implementations of Sort-Tile-Recursive. Only the CPU implementation uses the layouts from the aforementioned layout algorithm for bulk loading, while the CUDA implementation only supports fanouts and aR-tree sizes that are powers of two with a specific layout.

The multi-threaded implementation of Sort-Tile-Recursive is not used by the multi-threaded implementation of the Block-based Algorithm, but we describe it anyway due to its relevance to the Distance-First Algorithm. Assuming aR-trees have not been constructed prior to the join, the Distance-First Algorithm must perform two large bulk loads which can be parallelized.

6.3.1 Spatial partitioning

The spatial partitioning problem is the following: Given a set of spatial objects, we would like to partitions the objects into spatially separated groups of objects. Each object must be spatially close to objects in the same group, but spatially distant from objects in other groups. This problem must be solved as part of the Sort-Tile-Recursive algorithm to partition sets of R-tree objects into leaf nodes and to partition sets of leaf nodes into inner nodes. We are particularly interested in a solution that can distribute MBBs into fixed-size groups of r according to the fanout of the R-tree to match the linearized layout.

The Tile Partition Algorithm takes an array of d-dimensional points and partitions all the points into partitions of size r by partitioning the space into *tiles* where each tile contains a group of r distinct points. It is an important building block for the Sort-Tile-Recursive implementation. By representing an array of node records as an array of simple d-dimensional points, the algorithm can be used to arrange the records into r-sized spatially separated groups of records which can be turned into spatially separated nodes. In our implementation, the bottom point of the MBB of each record is used to represent each record as a point for the Tile Partition Algorithm.

The algorithm works by iteratively dividing the input and space into smaller segments, considering one spatial dimension at a time. In the two-dimensional case, the algorithm creates two-dimensional tiles by first dividing the input into slice segments, then dividing each slice segment into tiles. In the three-dimensional case, the algorithm creates three-dimensional blocks by first dividing the input into slab segments, then dividing each slab segment into pillar segments, then dividing each pillar segment into block segments.

s is the *divisor*, which is the amount of spatially separated sub-segments that the algorithm will divide each segment into for each dimension. It is the smallest integer s satisfying $rs^d \geq |S|$. In each iteration, the algorithm will attempt to divide each segment into s spatially separated segments by sorting by the next dimension.

Algorithm 6.2 Tile Partition Algorithm. S is a list of d-dimensional points, r is the block size, and d is the number of dimensions.

```
1: function TilePartition($S, r, d$)
2:     Initialize K as temporary list of $|S|$ items
3:     $s \leftarrow \left\lceil \sqrt[d]{\frac{|S|}{r}} \right\rceil$
4:     $l \leftarrow rs^d$
5:     for $i \leftarrow 0, d-1$ do
6:         $S \leftarrow$ SegmentedBlockSortByDim(S, l, d)
7:         $l \leftarrow \frac{l}{s}$
8:     end for
9:     return $S$
10: end function
```

The last or only segment may be smaller than the current segment size l, which is initialized as rs^d.

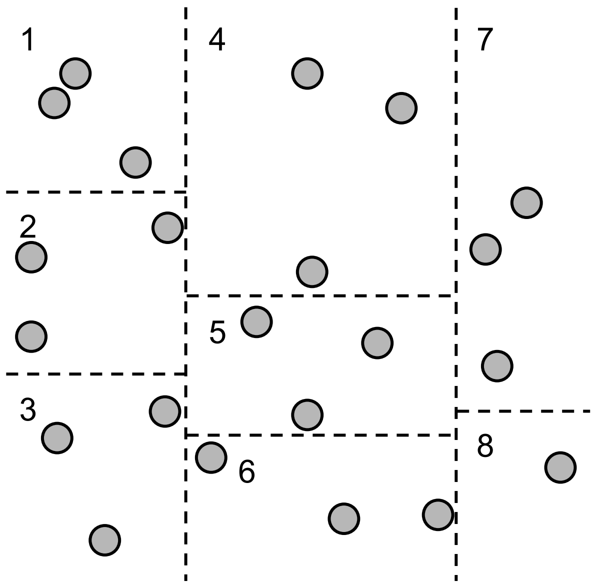

Figure 6.2: Tile partitioning

Figure 6.2 demonstrates an application of tile partitioning for 25 two-dimensional points partitioned into groups of $r = 3$. It uses a divisor of 3. The points are first partitioned into slabs of $3^2 = 9$ points each by x coordinate (vertical lines left to right), then each slab is further partitioned into tiles of 3 by y coordinate (horizontal lines top to bottom). All but the final slab and the final tile are fully packed with points.

Single-threaded implementation

The single-threaded implementation of the Tile Partition Algorithm simply repeatedly calls the single-threaded segmented implementation of the block sorting algorithm with decreasing block sizes and segment sizes. Each time the sorting implementation is called, it is given a comparator that compares records by the bottom coordinate in the current dimension.

Multi-threaded implementation

The tile partitioning implementation is expected to be called multiple times, and the segmented block sorting algorithm it relies will also be invoked multiple times per call. To avoid the overhead of creating and freeing threads for each call to the parallel segmented block sorting algorithm, both the segmented block sorting algorithm and the tile partitioning algorithm are designed to work with a thread pool to keep a fixed amount of threads alive. The thread pool is a fork-join pool, meaning that multiple threads are utilized by submitting tasks to the fork-join pool that each thread will execute in parallel.

CUDA implementation

Unlike the CPU implementation which uses a specialized block sorting algorithm, the CUDA implementation uses a regular segmented sorting algorithm. The CUDA implementation of tile sorting is based on a parallel segmented bitonic merge sort designed to operate in shared memory with one thread per input. This limits the size of the input, but ensures that items can move efficiently in the sorting network without accessing device memory.

6.3.2 Parent creation

The R-tree is built from the bottom up. Once the R-tree node records on level i have been partitioned into tiles of size r, they are ready to be assembled into inner node records for level $i + 1$. Parent creation takes an input segment of R-tree node records and produces an output segment of inner node records. It does not sort the input records, but rather reduces each block of records from a sorted array into one parent record per block. It calculates and stores the aggregate scores and the aggregate MBBs as an inner node record in the output segment. The score aggregation function is used to calculate an aggregate score for each node.

Algorithm 6.3 Create Parents Algorithm. S is a segment of the R-tree record array, r is the block size and γ is the score aggregation function.

```
1: function CreateParents($S, r, \gamma$)
2:     Initialize $O$ as list of $\lceil |S|/r \rceil$ items
3:     for all $i \in 0, 1, \ldots, \lceil |S|/r \rceil - 1$ do
4:         $j \leftarrow ir$
5:         $k \leftarrow$ Min($j + r, |S|$)
6:         $O_i$.mbb $\leftarrow$ Reduce(Mbb, $\{S_j.\text{mbb}, S_{j+1}.\text{mbb}, \ldots, S_{k-1}.\text{mbb}\}$)
7:         $O_i$.score $\leftarrow$ Reduce($\gamma$, $\{S_j.\text{score}, S_{j+1}.\text{score}, \ldots, S_{k-1}.\text{score}\}$)
8:         $O_i$.idx $\leftarrow j$
9:     end for
10:    return $O$
11: end function
```

The algorithm is based on the reduce primitive, represented here as a higher-order function.

The parent creation problem is embarrassingly parallel. There is no dependency between each block of r input items and each output item. Not only can the algorithm be parallelized per output item, it can also be parallelized for each input item with parallel reductions.

CPU implementation

The CPU implementation operates generically using an input iterator for S and output iterator for O. Using memory pointers as iterators, the algorithm can directly read and write segments in the R-tree record array. S points to the first record of the input segment, and O points to the first record of the output segment.

To parallelize the algorithm, output items are computed and written in parallel. Generally, there will be many more output items than there are threads, and the fanout will be too low to benefit from parallel reductions on the CPU. Therefore, the parallel CPU implementation will only parallelize the for loop and not the reduction. Given p threads, the output is divided into p evenly sized partitions that each thread can compute in parallel.

CUDA implementation

The CUDA implementation can exploit the high parallelism, not only by assigning a thread to each output item, but rather by assigning a thread to each input item. The CUDA implementation restricts the fanout r to a power of two no larger than 32. This enables the use of warp-level parallel reduction, which is used with one item per thread to compute the MBBs and aggregate scores without any thread block synchronization.

6.3.3 Sort-Tile-Recursive

Finally, the Sort-Tile-Recursive algorithm can be fully described. Given a list of objects I, each with an MBB, a score and some data value, the algorithm produces an array of aR-tree records according to the layout L containing all the input objects as leaf node records. First, the STR algorithm copies each input object as a leaf node record in the bottom segment of the aR-tree record array. For all but the last segment (the root level), the algorithm partitions the node records in the current segment into tiles using tile partitioning, then builds the next segment of node records for the next iteration. The last level is the root level and, therefore requires no tile partitioning, nor has any level above it to be built, so the algorithm terminates once the root level is built.

Single-threaded implementation

The single-threaded implementation of Sort-Tile-Recursive is quite close to the described algorithms. Given a pre-allocated array of aR-tree records, it starts by copying all the input item as aR-tree leaf records and inserting them into the bottom segment. It operates in-place in the aR-tree record array by carrying out tile partitioning on each segment in-place in the aR-tree record array. Parents are created using a simple nested loop.

Algorithm 6.4 Sort-Tile-Recursive. L is the layout of the R-tree, I is the list of objects in the R-tree, r is the block size, d is the number of dimensions and γ is the score aggregate function.

```
1: function SortTileRecursive($L, I, r, d, \gamma$)
2:     Initialize $R$ as list of $L_0$.offset + $L_0$.size items
3:     for all $i \in 0, 1, \ldots, |I| - 1$ do
4:         $R_{L_0.\text{offset}+i}$.mbb = $I_i$.mbb
5:         $R_{L_0.\text{offset}+i}$.score = $I_i$.score
6:         $R_{L_0.\text{offset}+i}$.data = $I_i$.data
7:     end for
8:     for $i \leftarrow 0,\ |L| - 1$ do
9:         $j_0 \leftarrow L_i$.offset
10:        $k_0 \leftarrow j_0 + L_i$.size $- 1$
11:        $j_1 \leftarrow L_{i+1}$.offset
12:        $k_1 \leftarrow j_1 + L_{i+1}$.size $- 1$
13:        $R_{j_0,j_0+1,\ldots,k_0} \leftarrow$ TilePartition($R_{j_0,j_0+1,\ldots,k_0}, r, d$)
14:        $R_{j_1,j_1+1,\ldots,k_1} \leftarrow$ CreateParents($R_{j_0,j_0+1,\ldots,k_0}, r, \gamma$)
15:    end for
16:    return $R$
17: end function
```

Multi-threaded implementation

Each step in Sort-Tile-Recursive is inherently sequential, as each step depends on the results of the previous step, but each step can be parallelized. A fork-join pool is employed to utilize a fixed amount of threads when calling the parallel implementations of TilePartition and CreateParents. The parallelism of

The multi-threaded implementation of Sort-Tile-Recursive, however, is not used by the multi-threaded implementation of the Block-based Algorithm. The Block-based Algorithm will instead attempt to perform multiple bulk loads in parallel using the single-threaded implementation. The usefulness of the multi-threaded implementation is instead that it may be used by the Distance-First Algorithm to speed up bulk loading the two aR-trees prior to the join.

CUDA implementation

The CUDA implementation of Sort-Tile-Recursive is based on a *workspace*, which is a large array of aR-tree records contained in shared memory. The workspace will be used to carry out sorting in shared memory and has to be large enough to fit at least the bottom segment (the largest segment) of the aR-tree record array that it creates. To start, the input items are read from global memory into the workspace as aR-tree leaf node records. The items in the workspace are then sorted using tile partitioning, then written out to global memory as the bottom segment. The records in the workspace are then assembled into nodes by creating an inner node record for each block of r records in the workspace. The inner node records are written back into the workspace, overwriting the previous segment of records, creating a smaller array that is ready to be sorted like the previous one. This is repeated until the array of records in the workspace is small enough to be the root segment.

The segments of the aR-tree are written to a pre-allocated array of aR-tree records in the order they are produced. The bottom segment is sorted and written first, and the root segment is written last without any sorting. Instead of using a pre-calculated layout, each segment is written contiguously into the array, calculating the offsets and sizes of the segments on the fly.

The design of the CUDA implementation restricts both the size of the input and the fanout. Both the size of the input and the fanout must be powers of two. To create the inner node records using warp-level reductions, the fanout can also be no larger than 32. The maximum size of the input is determined by the availability of shared memory and the size of each record in memory. The size of a record depends on the dimensionality and the data types used, while the availability of shared memory depends on a number of parameters. The workspace can generally be expected to fit at most 1024 records, creating aR-trees containing up to 1024 objects.

6.4 Ranked aR-tree join

Given two aR-trees L and R, the Ranked aR-tree Join Algorithm joins the aR-trees and outputs at most k score-data tuples with the best scores according to the spatial join predicate ϕ and the aggregate score function γ. For the purposes of using ranked aR-tree joins as part of the Block-based Algorithm, instead of defining the algorithm as returning an ordered list of top-k items, we define the algorithm to output its results into the min-heap O with a fixed capacity for at most k results, with a given score threshold θ. For a normal top-k join on two aR-trees, the algorithm is given an empty min-heap for results and a score threshold initialized to the smallest possible value of the score data type such as $-\infty$. By using a min-heap to store the results, they can be retrieved in order after a ranked aR-tree join by popping each item in the output heap in sequence.

In situations where a ranked aR-tree join is only part of a larger top-k spatial join such as in the Block-based Algorithm, the input to the Ranked aR-tree join Algorithm may differ. The algorithm may be given a heap that contains prior results from other joins, and the score threshold may be initialized to some other value as a result. The caller of the Ranked aR-tree join Algorithm may also know a score threshold greater than the k^{th} greatest score in the output heap. Prior candidate results and greater initial score thresholds cause the the algorithm to prune pairs of objects in the search space more aggressively, reducing the time it takes for the algorithm to terminate.

The algorithm operates on a priority queue. Each item in the queue refers to two intersecting nodes, one from L and the other from R. The priority queue is initialized with references to the root nodes of L and R. In each iteration, the top item of the queue is dequeued, then all records of the node of A and B are combined. If the pair of nodes are inner nodes, the pairs of children of N_L and N_R whose MBBs satisfy the spatial join predicate and whose aggregate scores are above the score threshold are placed into the priority queue. If the pair of nodes are leaf nodes, valid pairs of input items are instead inserted into the output min-heap.

The algorithm eagerly places results into the output heap and continuously updates the score threshold θ until the algorithm may terminate. Before the algorithm terminates, all results in the output are called *candidate results*, because they may be replaced later by the pop-insert operation that is used when the output heap is at capacity. Until the output heap contains k candidate results, θ remains at its initial value. The score threshold θ then is continuously updated to represent the k^{th} greatest score seen so far when new candidate results are found. When there are no items in the priority queue with a score above θ, the join can no longer produce any candidate results with a score above the score threshold, so the algorithm terminates. The candidate results in the output heap then become the true results of the ranked aR-tree join.

6.4.1 Single-threaded implementation

The single-threaded implementation is based on the priority queue in the C++ standard template library and a binary heap for its outputs. Unlike the algorithm that was described, the implementation actually supports joining aR-trees of different heights. Items in the priority queue use a $\{score, level_L, idx_L, level_R, idx_R\}$ struct, where each pair of level and idx fields identifies an aR-tree node record by its level and its position in the linearized aR-tree node record array. Items in the priority queue may therefore refer to both nodes and input items.

When an item is retrieved from the priority queue, its score is compared with the score threshold to determine if the function should terminate. The implementation will first fetch both records from the aR-tree record arrays, then determine two arrays of records to join based on the records that were fetched. If a node retrieved from the queue is an inner node record (as determined by the level field), its block of child records is used as an array for the join. If a node retrieved from the queue is otherwise a leaf node record, the record itself is used as the only entry of an array for the join. When the two arrays have been determined, a nested loop is used to join pairs of records satisfying the spatial join predicate with a score above the score threshold. If either or both records in a joined pair are inner node records, a pair of references to records is inserted into the priority queue. Only if both records are leaf node records will items be inserted into the output heap. The items in the output heap use a $\{score, data_L, data_R\}$ struct. The output heap item struct is essentially the same struct that is used for the top-k spatial join output.

6.4.2 Multi-threaded implementation

The Ranked aR-tree Join Algorithm is not very parallelizable on the CPU, and is therefore not a target for parallelization. The issue is that the algorithm is based entirely on many fine-grained tasks that have to be inserted into and popped from the priority queue, which is difficult to do concurrently. Concurrent priority queue and heap data structures exist, but they are complex and would likely incur a significant synchronization overhead due to the sheer volume of operations.

Parallelization can instead be achieved by performing multiple aR-tree joins using the single-threaded implementation in parallel. This is a viable option for the Block-based Algorithm, which consists of a series of aR-tree joins.

6.4.3 CUDA implementation

The CUDA implementation uses only 32 threads, which is the exact amount of threads required to operate the heap implementation that it is based on for its

Algorithm 6.5 Ranked aR-tree Join Algorithm. O is the output min-heap for candidate results, k is the desired amount of results, R_L is the root node record of the first aR-tree, R_R is the root node record of the second aR-tree, θ is the score threshold, ϕ is the spatial join predicate used to join the R-tree nodes and γ is the score aggregate function.

```
1: function RankedRTreeJoin($O, k, R_L, R_R, \theta, \gamma, \phi$)
2:     Initialize $Q$ as a priority queue of node record pairs ordered by $\gamma$
3:     if $\phi(R_L.\text{mbb}, R_R.\text{mbb})$ then
4:         Enqueue($Q, (R_L, R_R)$)
5:     end if
6:     while $Q$ is not empty do
7:         $(N_L, N_R) \leftarrow$ Dequeue($Q$)
8:         if $\gamma(N_L.\text{score}, N_R.\text{score}) \leq \theta$ then
9:             return                ▷ No more candidate results with sufficient score
10:        end if
11:        if $N_L$ and $N_R$ are inner nodes then
12:            for all $(C_L, C_R) \in N_L \times N_R$ do
13:                if $\phi(C_L.\text{mbb}, C_R.\text{mbb})$ and $\gamma(C_L.\text{score}, C_R.\text{score}) > \theta$ then
14:                    Enqueue($Q, (C_L, C_R)$)
15:                end if
16:            end for
17:        else
18:            for all $(C_L, C_R) \in N_L \times N_R$ do
19:                $s \leftarrow \gamma(C_L.\text{score}, C_R.\text{score})$
20:                if $\phi(C_L.\text{mbb}, C_R.\text{mbb})$ and $s > \theta$ then
21:                    if $|O| < k$ then
22:                        Insert($O, s, (C_L.\text{data}, C_R.\text{data})$)
23:                    else
24:                        PopInsert($O, s, (C_L.\text{data}, C_R.\text{data})$)
25:                    end if
26:                    if $|O| = k$ then
27:                        $\theta \leftarrow$ Peek(O).score
28:                    end if
29:                end if
30:            end for
31:        end if
32:    end while
33: end function
```

priority queue and output heap. The heap implementation is based on the parallel heap implementation by Crosetto [7]. It has been enhanced to support generic keys, values and comparators with a few optimizations. Additionally, our implementation resolves a critical issue with the implementation of the heap insert procedure that had gone unnoticed in the original implementation.

The size of the aR-tree inputs is restricted so that both aR-trees must be constructed with the same amount of objects and the same fanout, and both the amount of objects in each aR-tree and the fanout must be a power of two. Among other things, the limitations guarantee that the aR-trees are equally tall and equally wide on each level, and that all blocks in the aR-tree record array except the root node block are fully packed. Aside from this, given sufficient space for the priority queue, the implementation can support arbitrarily large aR-trees.

The priority queue has a fixed but adjustable capacity. An aR-tree join may fail if the priority queue overflows, in which case the execution is terminated and the error is reported to the caller. The caller must therefore estimate how much capacity is required to process the join, which increases with the height of the aR-trees and the amount of pairs of objects that satisfy the spatial join predicate, and also depends on the spatial distribution of scores. Using a capacity that is unnecessarily large results in allocation of unused device memory.

The items in the priority queue use a $\{level, idx_L, idx_R\}$ struct to refer to a pair of aR-tree nodes. The score of each pair of nodes is stored separately as the key in the heap data structure. Each idx field points to a fully packed block of records, each in its respective aR-tree record array. Because the aR-trees are equally tall, the level field stores the shared level of both nodes. Unlike all other nodes, the root nodes may have non-packed blocks in the aR-tree record array, and can therefore not be validly represented as a pair in the priority queue. The first iteration is therefore handled as a special case with non-full nodes.

After dequeuing a pair of nodes to join, the CUDA implementation first reads the full blocks of records of N_L and N_R from global memory into shared memory using the idx_L and idx_R values. Pre-fetching these records into shared memory ensures that device memory does not have to be repeatedly accessed to retrieve the same records from the two blocks when joining them. The shared memory requirements are proportional to the size of each block, which is the size of a record multiplied by the fanout, which is generally expected to be small.

To use the warp-level parallelism to its advantage, the CUDA implementation divides the input of the inner for-all loops into warp-sized blocks and uses each thread to evaluate aggregate scores and spatial join predicates in parallel. Anything that should be inserted into either the priority queue or the output heap is first written and compacted into an array in shared memory. Then all 32 threads cooperate to sequentially insert each item from the shared array into the priority queue or output heap.

The CUDA implementation of the Ranked aR-tree Join Algorithm is not expected to achieve speedups over the CPU implementations, but speedups may be achieved for massively parallel execution on multiple inputs. The somewhat random device memory access patterns to heap nodes and aR-tree nodes is expected to limit the performance of the implementation. Most importantly, the execution on the GPU should at least exhibit comparable performance to the CPU so that executing the Ranked aR-tree Join Algorithm on the GPU is a viable choice. If the implementation can execute on the GPU, it may run in parallel on multiple inputs to parallelize the execution to a much greater degree than what can be achieved on multi-core CPUs.

6.5 Block-based Algorithm

The implementation of the Block-based Algorithm resembles a Pull/Bound Rank Join, except that input items are pulled and joined with items pulled from the other input in units of fixed-size blocks. The inputs are two lists of input items L and R, a score aggregate function γ, two input-specific score aggregate functions γ_L and γ_R, and the spatial join predicate ϕ. The input-specific score aggregate functions are generally both MAX, but may be different for different orders and more complex score variations of γ. Similarly to the Ranked aR-tree Join Algorithm, we define the Block-based Algorithm as outputting its results into the min-heap O with a fixed capacity for at most k results, with a given score threshold θ.

In each iteration, the block-based algorithm determines which input to pull a block from and which input to probe in order to terminate as soon as possible. When there are multiple options of blocks to pull, our chosen strategy is to greedily maximize the upper score bound of the output items that can be produced from the block. The upper score bound of is the best score that can be achieved by joining all items in a block with all items in the entire other input. According to the theory of ranked joins using monotone aggregate score functions, it can be computed by applying the aggregate score function γ to the best score in the block and the best score in the entire other input. Prioritizing blocks by the upper score bound should generally result in the best items being found as quickly as possible and with as little work as possible.

A newly pulled block must be joined with each previously pulled block of the probed input, which is carried out using aR-tree joins. When a block of input items is pulled, an aR-tree is bulk loaded from the full block of items. A series of aR-tree joins is then carried out to join the blocks, placing candidate results into the output heap O. An important observation is that the upper score bound for the results of joining two blocks can be calculated, and that the upper score bound decreases with each block that is accessed from the probed input. Therefore, if blocks are accessed in the order they were pulled, the series of ranked aR-tree joins may terminate when the first block that gives an upper score bound below θ is encountered.

The Block-based Algorithm gathers candidate top-k results in the output heap O and continuously updates the score threshold θ just like the Block-based Algorithm. When enough candidate results have been gathered, the upper score bound is given the worst score value of the candidate result. When there are no more blocks to pull that give an upper score bound above θ, the algorithm may terminate.

Prior to joining the blocks in the Block-based Algorithm, each input is sorted so that blocks can be pulled in descending order of score. This can be carried out using block sorts with a slight adjustment: Because the algorithm needs to know the best score of a block, the item with the best score of each block must be placed as the first item in its block. This property is already satisfied by a full sort, but not for block sorts.

Algorithm 6.6 Block-based Algorithm. O is the output min-heap for candidate results, k is the amount of items to return, $|B|$ is the block size, r is the fanout, L and R are the spatial join inputs, θ is the score threshold, γ is the score aggregate function, γ_L and γ_R are score aggregate functions for their respective inputs and ϕ is the spatial join predicate.

```
 1: function BlockBasedRankedJoin($O, k, |B|, r, L, R, \theta, \gamma, \gamma_L, \gamma_R, \phi$)
 2:     BlockSort($L, |B|$)
 3:     BlockSort($R, |B|$)
 4:     while more blocks of objects exist in $L$ and $R$ do
 5:         $i \leftarrow$ next input to pull a block from ($L$ or $R$)
 6:         $j \leftarrow$ the input to probe ($L$ or $R$)
 7:         $B_i \leftarrow$ PullBlock($i, |B|$)
 8:         $T_i \leftarrow$ SortTileRecursive($B_i, r, \gamma_r, k$)
 9:         Initialize $V$ as list
10:         for all $T_j \in j$ do
11:             RankedRTreeJoin($O, k, T_L, T_R, \theta, \gamma, \phi$)
12:         end for
13:         if $|O| = k$ then
14:             $\theta \leftarrow$ Peek(O).score
15:         end if
16:         $score_{\max} \leftarrow \max(\gamma(L_{first}.\text{score}, R_{last}.\text{score}), \gamma(L_{last}.\text{score}, R_{first}.\text{score}))$
17:         if $score_{\max} \leq \theta$ then
18:             return                          ▷ No more items with sufficient score
19:         end if
20:     end while
21: end function
```

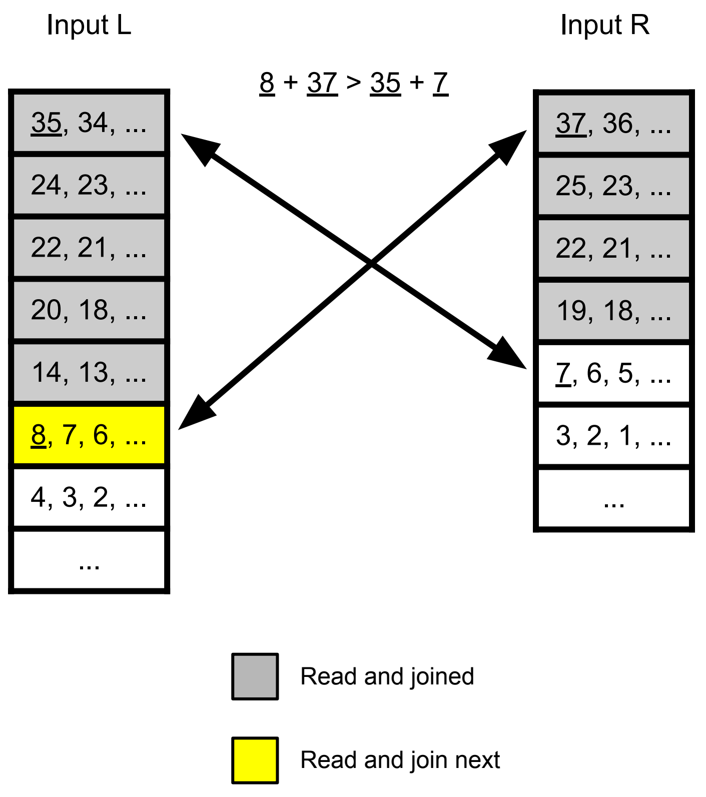

Figure 6.3: Block-based Algorithm block selection

6.5.1 Single-threaded implementation

To start, the single-threaded implementation sorts each input by score using the single-threaded block sorting algorithm. In addition, for each block in each input, the item with the best score in the block is placed as the first item of the block. This ensures that the upper score bound of any pair of blocks can be determined by checking the first item of each block. The output is accumulated in a binary min-heap with capacity for k items, which is passed by reference to each ranked aR-tree join. aR-tree record arrays are allocated dynamically each time an aR-tree is bulk loaded.

6.5.2 Multi-threaded implementation

The multi-threaded implementation of the Block-based Algorithm starts with a multi-threaded block sort for each input. The inputs are sorted in sequence, but each sort is carried out using the same multi-threaded block sort.

Due to its asynchronous nature, the multi-threaded implementation of the Block-based Algorithm distinguishes the *allocation* and the *initialization* of input blocks. When a block is *allocated*, the items in the block either have been bulk loaded into an aR-tree or are currently being bulk loaded by a thread. When a block is *initialized*, the items in the block have been bulk loaded into an aR-tree that is ready to be joined with other aR-trees. Blocks are allocated in the same order that they would be pulled in the single-threaded implementation, but they are initialized asynchronously.

For each input, pointers to bulk loaded R-trees are located in a pre-allocated array, but the aR-trees are allocated dynamically as they are bulk loaded. The state to keep track of which blocks are allocated and which blocks are initialized is represented as a simple integer counter for the number of blocks that are allocated and a set of uninitialized allocated blocks. Blocks are always allocated in order, therefore allocating a block is as simple as finding the next unallocated block and incrementing the allocated block counter. Additionally, since a newly allocated block is not immediately initialized, the block is immediately added to the set of uninitialized allocated blocks. When a block is initialized, it is removed from the set of uninitialized allocated blocks. This state representation is designed to be lightweight and to be easy to copy. Generally, the total amount of uninitialized blocks will be much lower than the amount of initialized blocks, and at most will be equal to the amount of threads, therefore the size of the set of uninitialized blocks is very limited.

Each thread repeatedly queries a shared data structure to issue asynchronous tasks to process until the result of the top-k spatial join is fully determined. A thread may allocate a block, which issues a *block initialization task* which must be processed by bulk loading the items of the newly allocated block into an aR-tree. When the block initialization task is complete, the thread must submit its results, which issues a *block join task*, which must be processed by joining the newly initialized block with all currently initialized blocks from the other input. Every time a task is produced and the results of a task are submitted, the implementation enters a critical section that manages the state of the blocks of both inputs.

Because a block join task may be processed asynchronously with other block initialization tasks, additional blocks may be initialized by other threads while the block join task is being processed. To avoid duplicated results, the block join task is restricted to only join with the blocks that were initialized when the task was issued, and not with any blocks that become initialized during processing. If a thread T_1 is issued a block join task for an initialized block B_1 that should at some point be be joined with a block B_2, and a thread T_2 initializes B_2 while T_1 is working on its block join task, T_1 should not join B_1 and B_2 because T_2 will be given the responsibility of joining B_1 and B_2 when a block join task for B_2 is issued to T_2. To this end, a snapshot of the set of initialized blocks is captured each time a block join task is issued, and the block join task is processed by joining with the collection of blocks in this snapshot. Capturing the snapshot is cheap due to the ease of copying the lightweight state representation. Only the counter for the amount of allocated blocks and the set of uninitialized blocks has to be copied each time a snapshot is captured.

The result is accumulated using one shared output min-heap and score threshold, and a number of local output min-heaps and score thresholds. When a thread is issued a block join task, the thread copies the shared score threshold and initializes its own empty local output heap for the sequence of joins it has been tasked to do. When the task is complete, candidate results from the local output heap are moved in bulk to the shared output heap. The shared heap is not concurrent, it can only be modified in a critical section.

The shared score threshold value at the time a block join task is issued is copied can be used to restrict the search space for the task, but not completely. We know that the score threshold can only increase as the algorithm progresses, and it may do so while a block join task is being processed. We know that any candidate result that cannot be placed into the local heap due to the local score threshold cannot be placed into the shared heap either. Conversely, we do not know if a candidate result placed in the local heap should be placed into the shared heap until the shared threshold value can be read again. The local score threshold only helps restrict the search space, while it cannot determine an exact threshold. The local score threshold is used as an input to the ranked aR-tree joins.

Inserting items into the shared heap in bulk causes the increase of the shared threshold value to be more staggered than in the single-threaded implementation, which may decrease work efficiency, but keeps the communication cost low. When the threshold does not increase as fast as theoretically possible, more work may have to be carried out to determine the results because the search space cannot be pruned as efficiently. However, a heap that supports concurrent fine-grained operations would likely have a similar cost. Using a critical section keeps the communication cost quite low assuming the heaps are small enough.

To avoid the overhead of creating and freeing threads for each call to the parallel block sorting algorithm and to carry out the rest of the algorithm, the implementation is designed to work with a thread pool to keep a fixed amount of threads alive.

6.5.3 CUDA implementation

The CUDA implementation of the Block-based Algorithm is based on three kernels: The control kernel, the block initialization kernel and the block join kernel. The control kernel handles only the core logic of the Block-based Algorithm, invoking the other kernels to do the heavy work. It uses CUDA dynamic parallelism to invoke the block initialization kernel and the block join kernel. The block initialization kernel handles the bulk loading of input blocks into aR-trees. The block join kernel handles joining pairs of aR-trees into candidate results.

Some preparations have to be made prior to executing the CUDA implementation of the Block-based Algorithm. Each input first has to be sorted by score. Instead of using a specialized block sorting algorithm, the sorting is done on the GPU using a full radix sort that operates in global memory. The radix sort implementation is provided by CUB, a utility library for CUDA. The sorts are carried out in parallel using CUDA streams. A few chunks of global memory must also be allocated prior to the join to be used for aR-trees and temporary storage during the execution. A buffer must also be allocated in global memory prior for the implementation to write out the results.

The control kernel is executed as a single warp of 32 threads. It uses an output heap to gather candidate results until it terminates, when it writes each result to the buffer in global memory.

To parallelize the Block-based Algorithm, the control kernel will invoke other kernels to perform bulk loading in parallel and join pairs of blocks in parallel. The parallelism of the CUDA implementation is determined by the *block parallelism* parameter, which determines both how many blocks can be bulk loaded into aR-trees concurrently and how many sequences of aR-tree joins can be executed concurrently. In the while loop of the Block-Based Ranked Join Algorithm, instead of choosing a single block to pull, the implementation choses how many blocks to pull from each input. It uses the same principle of greedily maximizing the upper score bound when choosing. The control kernel can allocate as many blocks as the block parallelism allows in total in each iteration. When the allocation of blocks is determined, the control kernel allocates global memory space for aR-trees then launches the block initialization kernel.

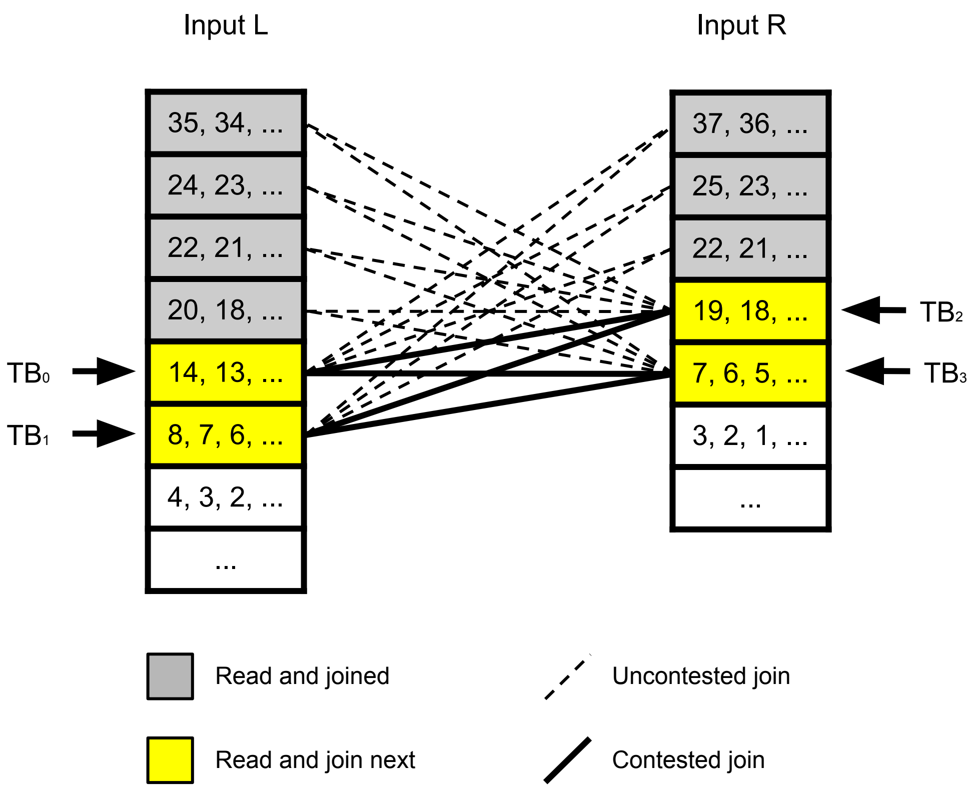

Figure 6.4: Block-based Algorithm multiple block selection

The block initialization kernel is launched so that there is one thread group per input block, where each thread block has as many threads as there are items in a block. Each thread block determines which block from which input to initialize, then calls the CUDA implementation of Sort-Tile-Recursive to write the aR-tree records into the array that was allocated by the control kernel.

When the block initialization kernel is complete, the control kernel launches the block join kernel to join each newly initialized block with all initialized blocks. The block join kernel is launched so that there is one thread group per newly initialized block with 32 threads per block, where each thread group is responsible for joining one newly initialized input block with each initialized block of the other input. Each thread block determines which initialized block is its own, then iterates over the initialized blocks of the other input performing a series of ranked aR-tree joins like the inner for loop of the algorithm. The block join kernel copies θ from the control algorithm as a local score threshold so that the series of aR-tree joins can start with more aggressive pruning. Using the same logic as the single-threaded and multi-threaded implementations, a thread group may stop processing aR-tree joins if it can determine that it cannot produce any more candidate results above its local score threshold. When the aR-tree joins of a thread block are complete, the thread block pops each item from its output heap and writes it to its own buffer in global memory. A thread block in the block join kernel may fail if it fails to process a join due to a priority queue overflow. Success/failure is reported to the control kernel by each thread group writing a status code to global memory on termination.

Each thread block in the block join kernel is tasked with joining one newly initialized input block with *all* initialized input blocks of the other input, which may include other newly initialized input blocks that other thread blocks are tasked with. This creates *contested joins*, which are joins between two newly initialized input blocks where two thread groups are tasked to perform the same join. To avoid performing a contested join twice, only the first thread that attempts it may perform the join. To determine a thread block should perform a contested join, atomic flags are allocated in global memory for each contested join. When encountering a contested join, the thread block performs an atomic Compare-And-Swap operation on the flag to simultaneously determine if the contested join should be performed and, if it should be performed, to signal that it cannot be performed by other threads following the operation.

When the block join kernel is complete, the control kernel starts reading the outputs of the block join kernel. Each thread block has written a sorted array of results into its own buffer in global memory. The result items from the block join kernel have to be evaluated for insertion into the output heap of the control kernel. The control kernel iterates over the results in descending order of score by concurrently iterating over each result array. When retrieving the next result, warp-level parallel reduction is used to find the best iterator, which is the iterator pointing to the item with the best score. The value pointed to by the best iterator is placed into the output heap, and the best iterator is then moved to the next position in its array. Because the results are iterated in descending order of score, the control kernel can stop reading the block join kernel results when it encounters the first item with a score below θ.

The BA may fail in one of three ways. It may fail if there is an error with the dynamic parallelism that causes any kernel to fail to launch, which can happen for various reasons. It may also fail if the block join kernel fails to process an aR-tree join due to priority queue overflow. The last failure condition is if it runs out of memory to allocate for bulk loading of aR trees. Success/failure of the control kernel is reported by writing a status code to global memory on termination.

The most important reason for using a control kernel instead of controlling the program from the host is that it can operate entirely within device memory. The control kernel performs many reads in global memory during the execution. Controlling the algorithm from the host would involve a number of transfers between host memory and device memory that would cause the execution to be slower.

Chapter 7

Experimental evaluation

7.1 Setup

All experiments were carried out on the same system. The CPU is 4-core Intel Core i5 6600K running at 3.50GHz with one thread per core. The GPU is an NVIDIA GeForce GTX 1080. The system has 16GB of RAM. The operating system is a 64-bit version of Microsoft Windows 10.

All implementations are written in C++. Single-threaded and multi-threaded implementations are compiled using Visual Studio, while CUDA implementations are compiled with NVCC using Visual Studio as the host compiler. Everything is compiled with the -O2 flag to optimize for speed, and CUDA code is compiled without the -G debug information flag to optimize the performance of kernel code.

7.2 Methodology

To measure the performance of the implementations, we measure the run time of experiments using the C++ high resolution clock API. The Visual Studio implementation of the high resolution clock measures time in units nanoseconds with a resolution below one nanosecond. Each experiment is a function that is given two input buffers, one output buffer and the parameters for a top-k spatial query on the inputs. The input buffers contain the input data in memory, while the output buffer is where the experiment is expected to write its results to. Prior to running an experiment, the current time is recorded as t_{start}. The experiment is then run immediately. When the experiment is complete, the current time is recorded again as t_{stop}. The run time of an experiment is measured as $t_{stop} - t_{start}$. Each experiment is repeated five times, and the average run time is recorded as a result.

We perform both CPU and CUDA experiments using the same experimental method. All input and output buffers reside in host memory prior to the experiment, so that each CUDA experiment must copy the inputs from host memory to device memory and the outputs back from device memory to host memory. We also use the same method to measure the run time of CUDA implementation. Because CUDA is largely an asynchronous API, CUDA has a system for timing events that is commonly used to measure the performance of CUDA applications. However, we intend to measure the performance of the CUDA implementation from the perspective of an application that keeps most of its logic and data on the host. To the host, the only time that matters is the time it takes for the implementation to place the output into the buffer so that it may be used in further processing.

To avoid processor cache behavior causing interference in the run time between experiments, the experimental evaluation makes an effort to flush processor caches between each experiment on the CPU. This is carried out by filling a 80MB buffer in memory with random values before each experiment. The same flushing is not used before CUDA experiments due to the processing taking place on the device and not on the host. All buffers in global memory used by a CUDA experiment are allocated each time the experiment is run, but it is uncertain whether or not different regions of device memory are allocated in each experiment. However, the majority of buffers in global memory are filled with data before any processing happens on the GPU, which should invalidate any caches. This means GPU cache behavior is unlikely to affect the run time of CUDA experiments.

The input data consists of two randomly generated data sets to join called L and R. When using the data sets for queries, the first 1048576 items, 32768 or 1024 items are used. The data items are uniformly distributed two-dimensional points with single-precision floating point coordinates between 0 and 1000. Each item has a 32-bit integer score between 0 and 100 and a 32-bit integer data value equal to its position in the dataset. With eight bytes of padding, each record has a size of 32 bytes.

Each combination of input sizes was used to create six possible pairs of inputs: $L_1 \times R_1, L_1 \times R_2, L_1 \times R_3, L_2 \times R_2, L_2 \times R_3$ and $L_3 \times R_3$. 3 queries were performed on each pair of inputs. Each query requests the pairs of points with the best score sums with a distance threshold of exactly one. The first query Q_1 requests the first 10000 items with the best scores, the second query Q_2 requests 1000, and the final query Q_3 requests 100.

For each query applied to each pair of inputs, we measure the run time of the single-threaded implementation, the multi-threaded implementation and the CUDA implementation of the Block-based Algorithm. To compare the different implementations, the fanout and block sizes of each experiment is manually tuned to optimize each run time.

For CUDA experiments, a trial run of each experiment must be performed before any measurements can be made. Otherwise, a significant run time penalty may be measured the first time an experiment is run. This is caused by CUDA performing just-in-time compilation of kernels to match the hardware architecture, which will affect run times the first time a kernel is launched.

For multi-threaded experiments, the same thread pool of 4 initialized (but sleeping prior to the experiment) threads is used for all experiments. This matches the amount of cores in the system, and is the amount of threads that performs best on this system.

To compare the different implementations in the different experiments, it is assumed that the parameters for the implementations can be automatically tuned based on the spatial distribution of the input data, the distribution of input data scores and the size of the inputs and outputs. Because the automatic tuning of the parameters has not yet been implemented, the parameters have been manually tuned to maximize the run time of each experiment. The results therefore do not represent the true performance of an application of the implementations, but serves as a way to compare the implementations and how they respond to different input sizes and output sizes. The parameters may be subject to overfitting for the particular dataset used in the experiments.

7.3 Results

Because the run times of experiments on L_1 and the rest are on different orders of magnitude, the run times are displayed in two figures. Figure 7.1 shows the run times of all experiments including input L_1, one of the two largest inputs, while figure 7.2 shows the run time of the remaining experiments.

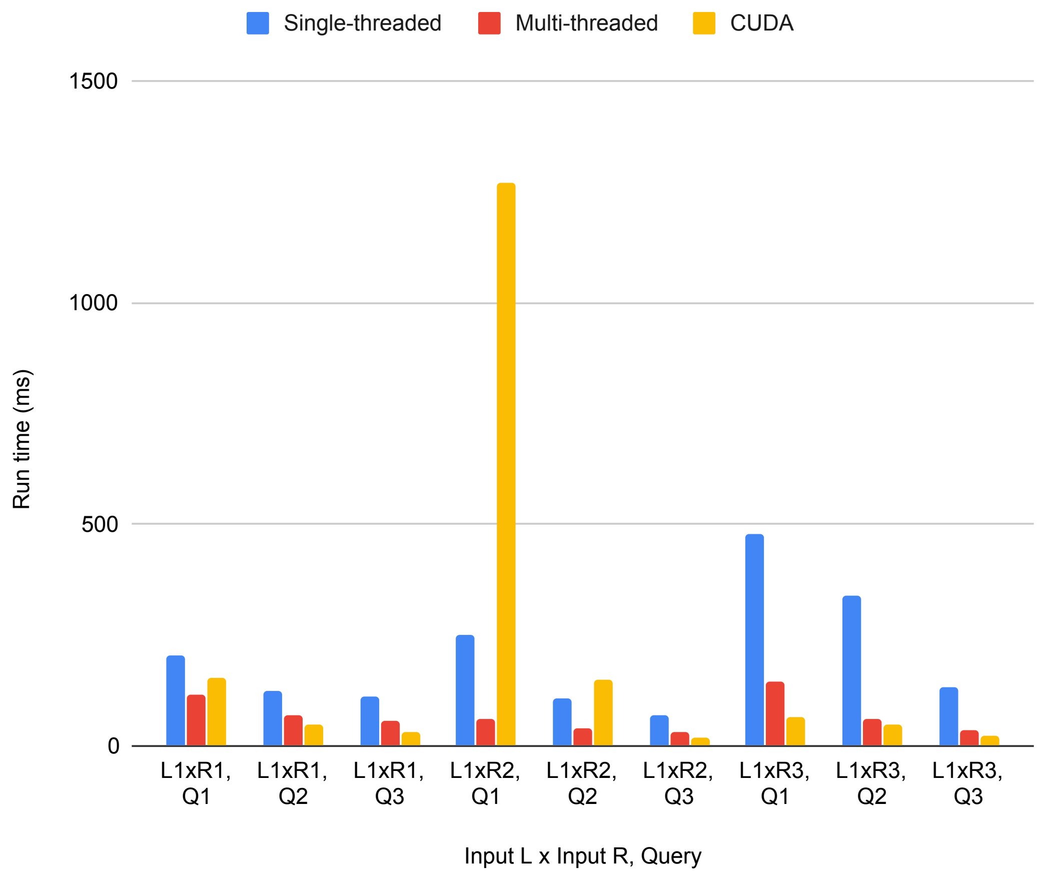

Figure 7.1: Run times of BA for large inputs

The run time of the single threaded implementation can be improved upon in all situations except for queries on L_3xR_3. The multi-threaded implementation has consistently better run time than the single-threaded implementation in all situations except for the smallest inputs, L_3xR_3. This is to be expected from the utilization of multiple processor cores. At a certain point, the cost of communication between the threads outweighs the gains of parallel processing, as can be seen for L_3xR_3. The CUDA implementation is most efficient in situations with large inputs and small outputs. This is likely due to the fact that the radix sort used on the inputs prior to the sort on the GPU is more efficient than the custom sorting implementation on the CPU above a certain size threshold. The process of initializing and joining

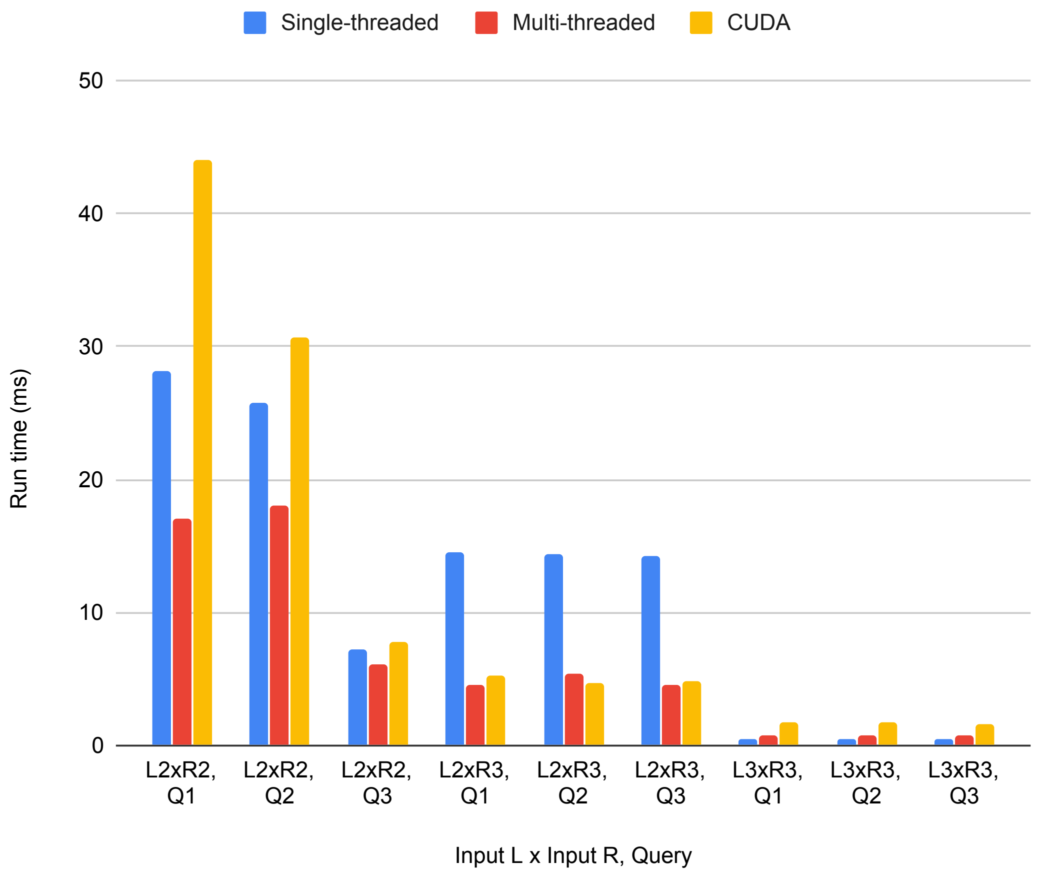

Figure 7.2: Run times of BA for small inputs

blocks in the CUDA implementation is likely also slower than on the multi-threaded implementation.

The output size is generally expected to affect the run time of all implementations in most situations, which is seen in varying patterns. For L_2xR_3 and L_3xR_3, the total amount of pairs of points satisfying the distance threshold is likely smaller than the output size, causing little to no pruning of the search space based on score, so that all pairs of blocks have to be joined regardless of output size.

7.3.1 Effect of input size skew

All implementations have noticeable issues with input size skew. The single-threaded implementation and CUDA implementation seem to have increased run times for pairs of inputs with skewed sizes. The join space of L_1xR_2, Q_1 is much smaller than the join space of L_1xR_1, Q_1, which should reduce the run time, but the run time is instead much larger. The multi-threaded implementation exhibits the expected effect on run time of reducing the size of the join space.

Input size skew coupled with large output sizes likely causes increasingly unbalanced workloads in the CUDA implementation. When one input is larger than the other, the Block-based Algorithm is much more likely to select a block from the largest input. In the likely event that it selects a block from the largest input, the sequence of joins to process following the initialization is small because there are fewer blocks to join with in the smallest input. In the less likely event that it selects a block from the smallest input, the sequence of joins to process following the initialization is large because there are more blocks to join with in the largest input. If the output size is large, the imbalance will grow over time and the processing of a sequence of joins is less likely to terminate early due to the threshold climbing more slowly.

Each kernel launch in the control kernel of the CUDA implementation must wait for all thread groups to terminate. If the workload is not balanced across the thread groups, the control kernel must wait for the slowest thread group to terminate. This is likely happening, not with the block initialization kernel, but with the block join kernel due to the aforementioned imbalance. Unlike the CUDA implementation, the multi-threaded implementation processes each block join task independently, so that no thread has to wait for another thread to complete a block initialization task, and therefore does not suffer from the same issue.

The effect of the input size skew of L_1xR_2 is extreme for the CUDA implementation, but the even greater skew of L_1xR_3 has no noticeable effect on the CUDA implementation. This is because the block size used for this experiment is equal to the size of R_3, causing the Block-based Algorithm to select the only block of the smallest input immediately.

It is unknown why the input size skew affects the single-threaded implementation as much as it does, because it cannot suffer from the workload imbalance problem that likely affects the CUDA implementation due to the sequential nature of the single-threaded implementation. It could be a general issue with the Block-based Algorithm. It might be caused by the heuristic chosen for choosing blocks, which would affect all implementations. The current heuristic attempts to maximize the upper score bound of candidate results that can be produced by selecting the block. The algorithm might not have to find the candidate results with the best scores first, and should perhaps focus on producing as many candidate results as possible to increase the score threshold more rapidly. This would put more pressure on the result heap, but might prune the search space more efficiently.

7.3.2 Effect of thread count

To determine the effect of thread count on the multi-threaded implementation, experiments were performed with varying thread counts on L_1xR_1, Q_1. The thread count determines the amount of threads used both for the sort prior to the join, and the join itself. More threads means more segments of the arrays can be sorted in parallel, and more blocks can be initialized and joined in parallel, but also means more communication between threads. Results are shown in figure 7.3. The thread count shows a noticeable but limited effect on the run time. Speedups are achieved between 2 and 5 threads, with the lowest run time at 4 threads. Each thread added before 4 decreases the run time, while each thread added past 4 mostly increases the run time.

The thread count has an expected effect on the run time. Adding more threads adds more contention to the critical sections, causing diminishing returns with each thread. The processor used to run the experiments has 4 cores, which likely explains the run time increasing when adding more threads than four. Adding more threads

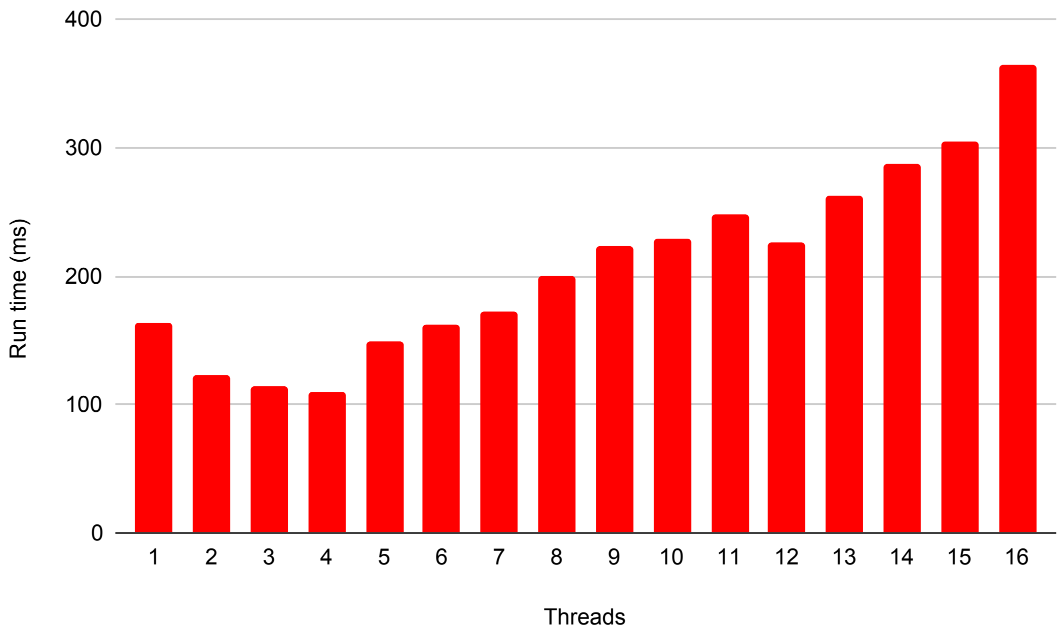

Figure 7.3: Run times of multi-threaded implementation for L_1xR_1, Q_1 by thread count

than cores may help in situations where threads have more complex cooperation or stalls due to I/O or page faults, but the experiment shows that having as many threads as there are cores is most efficient.

7.3.3 Effect of block parallelism

To determine the effect of block parallelism on the CUDA implementation, experiments were performed with varying block parallelism on L_1xR_1, Q_1. The block parallelism parameter does not affect the sorting prior to joining, but determines how many blocks can be initialized in parallel and how many series of aR-tree joins can be performed in parallel. Results are shown in figure 7.4. The block parallelism shows a noticeable effect on the run time. It trends towards lower run times with increasing block parallelism. A block parallelism greater than 32 could unfortunately not be achieved due to limitations with the current implementation.

The block parallelism has an expected effect on the run time. Increasing the block parallelism increases the amount of work required to produce and reduce block join kernel outputs, and also slightly reduces the efficiency of pruning the search space with the score threshold, causing diminishing returns, but not nearly as dramatically as for the thread count of the multi-threaded implementation. Increasing the block parallelism enables better distribution of the work, allowing the work to be distributed to all SMs, and allowing each SM to interleave the execution of thread blocks. The block parallelism cannot be increased indefinitely, but the trend in the data suggests that block parallelisms beyond 32 may give even lower run times in this particular experiment.

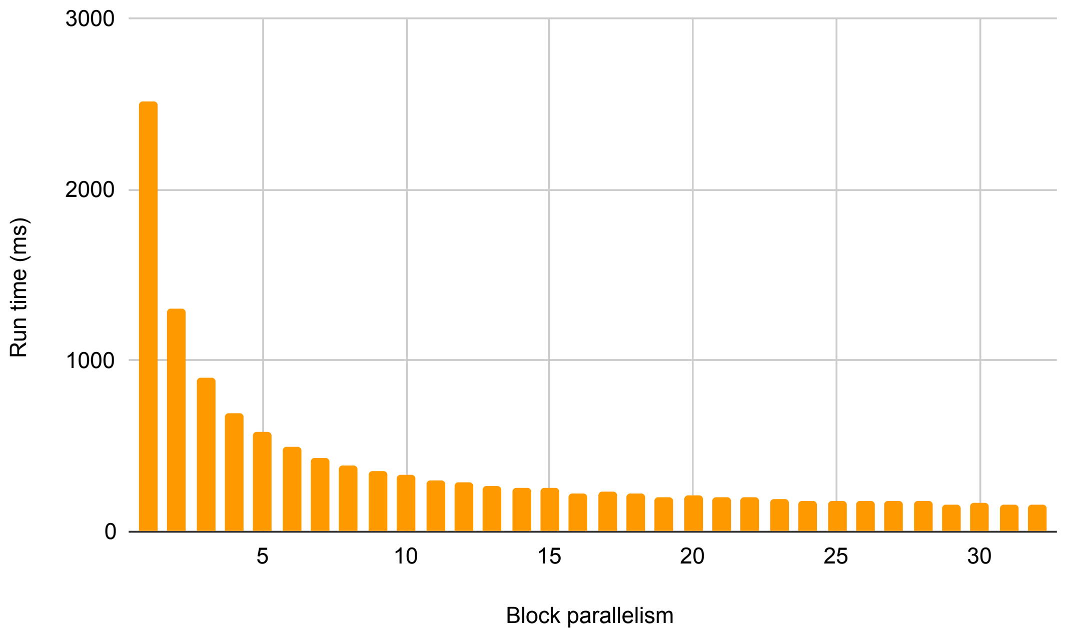

Figure 7.4: Run times of CUDA implementation for L_1xR_1, Q_1 by block parallelism

Chapter 8

Conclusions and future work

8.1 Conclusions

Speedups can be achieved for top-k spatial joins by parallelizing the Block-based algorithm using both multiple threads and GPGPU. Experimentation on the implementations that were developed shows that for inputs that are large enough, multi-threaded execution achieves significant speedups. Using GPGPU can be even more efficient for large inputs and smaller values of k, likely due to the more efficient GPGPU sorting of large arrays but slower execution of the rest of the Block-based Algorithm.

The Block-based Algorithm is proven to be effective for top-k spatial joins, and it can be parallelized by using a divide-and-conquer strategy for aR-tree joins. Instead of joining every pair of blocks in sequence, the joins can be processed in parallel and the results of each parallel join can be joined into a single heap. Increasing the parallelism reduces the work efficiency but increases the potential speedup. This methodology can be used to achieve speedups for both multi-threaded and GPGPU implementations of the Block-based Algorithm.

The Sort-Tile-Recursive R-tree bulk loading method can be performed on the GPU. Despite the steps of STR being inherently sequential, it parallelizes via parallel sorting algorithms and parallel reduction. By carrying out the sorting of Sort-Tile-Recursive in shared memory, we avoid the cost of accessing device memory.

A new relaxed variant of the sorting problem was identified and solved for the Block-based Algorithm and Sort-Tile-Recursive, which was used as an optimization for the single-threaded and multi-threaded implementations. We call this relaxed variant of the sorting problem "block sorting", which involves sorting the input into fixed-size blocks where the internal order of each block is relaxed. The sequential and parallel implementations of a specialized block sorting algorithm based on quick sort was shown to be more efficient for the block sorting problem than the sequential and parallel sorting functions in the C++ standard library. An optimized GPGPU solution to the block sorting problem was not implemented.

The use of a specially designed aR-tree memory layout and parallel heaps allows the execution of aR-tree joins on the GPU, which allows the full execution of the Block-based Algorithm on the GPU with comparable performance to the CPU. Executing the algorithm entirely on the GPU reduces the amount of transfers between device memory and host memory, whose bandwidth is often a bottleneck for GPGPU applications. The CUDA implementation appears to be largely memory bound. While the alignment of blocks of aR-tree records allows efficient use of device memory when performing aR-tree joins, the sheer amount of random accesses to the aligned blocks likely makes a significant negative performance impact. The parallel heap also uses device memory for its local memory, which likely makes a similar impact due to the frequency of heap operations to gather candidate results and to operate the priority queue, despite the efficient alignment of local memory. Still, the execution is reasonably efficient when parallelizing across multiple aR-tree joins.

The implementations have significant issues with skewed input sizes, likely due to issues with balancing the workload and the heuristic that was chosen for choosing blocks in the Block-based Algorithm. There is no reason for skewed input sizes to cause the top-k spatial join problem to become more difficult, because the amount of work should be proportional to the size of the join space, which is proportional to the size of both inputs multiplied. Additional experimentation would likely identify the cause of the issues.

The performance of the CUDA implementation is likely affected by its limitations. The greatest block size that could be achieved in the CUDA implementation is limited by the amount of records that can fit in shared memory, and the block parallelism is limited by the size of a warp. The block size limit is only imposed by the block initialization kernel which performs Sort-Tile-Recursive in shared memory, and not the block join kernel which can support much larger aR-trees by reading aR-tree nodes straight from global memory. Supporting larger aR-trees would require performing parts of Sort-Tile-Recursive in global memory instead of shared memory, which would reduce the efficiency of the block initialization kernel may require a different sorting algorithm, but still might improve the overall efficiency of the Block-based Algorithm. A block parallelism greater than 32 might further decrease the run time, and only requires slight adjustments to the control kernel.

8.2 Future work

The implementations that were created are not truly complete, and can likely be optimized and tweaked further with more experimentation and profiling to create more accurate data to compare the different methods. The current experimental data suggests that there are issues with the current implementations, especially for skewed input sizes. Given more development time, some of the limitations of the CUDA implementation can be relaxed by adding new and more complex features. The implementation of the control kernel has significant potential for optimization by improving global memory access patterns. There are a number of tools that can be used to analyze the performance of the code, particularly to analyze the performance of kernels.

This thesis has focused solely on the R-tree data structure for spatial indexing, while another avenue of research for top-k spatial joins is using other data structures such as the family of quadtree data structures. The R-tree is considered a data-driven spatial data structure because the way it partitions the space is dependent on the data. Unlike the R-tree, the quadtree is considered a space-driven spatial data structure because it uses a fixed division of the space. Because the division of the space is not dependent on the data, the process of indexing a block of spatial objects in a quadtree might behave differently in a way that is differently suited for GPGPU. However, for data whose indexed spatial attributes cannot be represented as a single point, the use of quadtrees would be more challenging. Input objects may have to be placed into multiple octree nodes, which introduces the problem of eliminating duplicates during spatial joins.

Another surprisingly simple way to achieve parallel top-k spatial joins might be using a parallel plane-sweep based join. While the plane sweep join algorithm cannot produce outputs in ranked order, a number of top-k joins could be performed in parallel by sorting the input data by one coordinate, then dividing the input space into segments and performing a top-k join on each segment. The top-k results could then be joined into a single top-k result. Alternatively, borrowing from the Block-based Algorithm, the input could be divided into blocks based on their individual scores, then pairs of blocks can be joined via plane-sweep joins.

Bibliography

[1] Erlend Åmdal. *Top-k Spatial Join on GPU.* Project report in TTM4502. Department of Information Security, Communication Technology, NTNU – Norwegian University of Science and Technology, Dec. 2019.

[2] Lars Arge et al. 'Scalable sweeping-based spatial join'. In: *VLDB.* Vol. 98. Citeseer. 1998, pp. 570–581.

[3] Norbert Beckmann et al. 'The R*-tree: an efficient and robust access method for points and rectangles'. In: *Acm Sigmod Record.* Vol. 19. 2. Acm. 1990, pp. 322–331.

[4] Panagiotis Bouros and Nikos Mamoulis. 'Spatial joins: what's next?' In: *SIGSPATIAL Special* 11.1 (2019), pp. 13–21.

[5] Thomas Brinkhoff, Hans-Peter Kriegel and Bernhard Seeger. *Efficient processing of spatial joins using R-trees.* Vol. 22. 2. ACM, 1993.

[6] NVIDIA Corporation. *CUDA C++ programming guide.* 2019. URL: `https://docs.nvidia.com/pdf/CUDA_C_Programming_Guide.pdf` (visited on 01/12/2019).

[7] Paolo G. Crosetto. *CUPQ: a CUDA implementation of a Priority Queue applied to the many-to-many shortest path problem.* Version 0.1. Dec. 2019. DOI: `10.5281/zenodo.3595244`. URL: `https://github.com/crosetto/cupq`.

[8] Antonin Guttman. *R-trees: A dynamic index structure for spatial searching.* Vol. 14. 2. ACM, 1984.

[9] Bingsheng He et al. 'Relational joins on graphics processors'. In: *Proceedings of the 2008 ACM SIGMOD international conference on Management of data.* ACM. 2008, pp. 511–524.

[10] Sangyong Hwang et al. 'Performance evaluation of main-memory R-tree variants'. In: *International Symposium on Spatial and Temporal Databases.* Springer. 2003, pp. 10–27.

[11] Ihab F Ilyas, Walid G Aref and Ahmed K Elmagarmid. 'Supporting top-k join queries in relational databases'. In: *The VLDB Journal—The International Journal on Very Large Data Bases* 13.3 (2004), pp. 207–221.

[12] Scott T Leutenegger, Mario A Lopez and Jeffrey Edgington. 'STR: A simple and efficient algorithm for R-tree packing'. In: *Proceedings 13th International Conference on Data Engineering.* IEEE. 1997, pp. 497–506.

[13] Zhisheng Li et al. 'Ir-tree: An efficient index for geographic document search'. In: *IEEE Transactions on Knowledge and Data Engineering* 23.4 (2010), pp. 585–599.

[14] Lijuan Luo, Martin DF Wong and Lance Leong. 'Parallel implementation of R-trees on the GPU'. In: *17th Asia and South Pacific Design Automation Conference*. IEEE. 2012, pp. 353–358.

[15] Rodrigo Paredes and Gonzalo Navarro. 'Optimal incremental sorting'. In: *2006 Proceedings of the Eighth Workshop on Algorithm Engineering and Experiments (ALENEX)*. SIAM. 2006, pp. 171–182.

[16] Shuyao Qi, Panagiotis Bouros and Nikos Mamoulis. 'Efficient Top-k Joins on Complex Data Types'. In: (2015).

[17] Shuyao Qi, Panagiotis Bouros and Nikos Mamoulis. 'Efficient top-k spatial distance joins'. In: *International Symposium on Spatial and Temporal Databases*. Springer. 2013, pp. 1–18.

[18] Karl Schnaitter and Neoklis Polyzotis. 'Optimal algorithms for evaluating rank joins in database systems'. In: *ACM Transactions on Database Systems (TODS)* 35.1 (2010), p. 6.

[19] Dhirendra Pratap Singh, Ishan Joshi and Jaytrilok Choudhary. 'Survey of GPU based sorting algorithms'. In: *International Journal of Parallel Programming* 46.6 (2018), pp. 1017–1034.

[20] Tongjai Yampaka and Prabhas Chongstitvatana. 'Spatial join with r-tree on graphics processing units'. In: *King Mongkut's University of Technology North Bangkok International Journal of Applied Science and Technology* 5.3 (2012), pp. 1–7.

[21] Simin You, Jianting Zhang 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*. ACM. 2013, pp. 23–31.

[22] Renwei Yu et al. 'Workload-balanced processing of top-K join queries on cluster architectures'. In: *tech. report ASUCIDSE-CSE-2010–001* (2010).