DEGREE PROJECT IN TECHNOLOGY,
FIRST CYCLE, 15 CREDITS
*STOCKHOLM, SWEDEN 2017*

# Improving the performance of GPU-accelerated spatial joins

**DUSAN VIKTOR HRSTIC**

**KTH ROYAL INSTITUTE OF TECHNOLOGY**
**SCHOOL OF INFORMATION AND COMMUNICATION TECHNOLOGY**

## Abstract

Data collisions have been widely studied by various fields of science and industry. Combing CPU and GPU for processing spatial joins has been broadly accepted due to the increased speed of computations. This should redirect efforts in GPGPU research from straightforward porting of applications to establishing principles and strategies that allow efficient mapping of computation to graphics hardware. As threads are executing instructions while using hardware resources that are available, impact of different thread organizations and their effect on spatial join performance is analyzed and examined in this report.

Having new perspectives and solutions to the problem of thread organization and warp scheduling may contribute more to encourage others to program on the GPU side. The aim with this project is to examine the impact of different thread organizations in spatial join processes. The relationship between the items inside datasets are examined by counting the number of collisions their join produce in order to understand how different approaches may have an influence on performance. Performance benchmarking, analysis and measuring of different approaches in thread organization are investigated and analyzed in this report in order to find the most time efficient solution which is the purpose of the conducted work.

This report shows the obtained results for the utilization of different thread techniques in order to optimize the computational speeds of the spatial join algorithms. There are two algorithms on the GPU, one implementing thread techniques and the other non-optimizing solution. The GPU times are compared with the execution times on the CPU and the GPU implementations are verified by observing the collision counters that are matching with all of the collision counters from the CPU counterpart.

In the analysis part of this report the the implementations are discussed and compared to each other. It has shown that the difference between algorithm implementing thread techniques and the non-optimizing one lies around 80% in favour of the algorithm implementing thread techniques and it is also around 56 times faster then the spatial joins on the CPU.



## Keywords

GPU-acceleration, spatial joins, threads optimizations, warp scheduling

## Sammanfattning

Datakollisioner har studerats i stor utsträckning i olika områden inom vetenskap och industri. Att kombinera CPU och GPU för bearbetning av rumsliga föreningar har godtagits på grund av bättre prestanda. Detta bör omdirigera insatser i GPGPU-forskning från en enkel portning av applikationer till fastställande av principer och strategier som möjliggör en effektiv användning av grafikhårdvara. Eftersom trådar som exekverar instruktioner använder sig av hårdvaruresurser, förekommer olika effekter beroende på olika trådorganisationer. Deras påverkan på prestanda av rumsliga föreningar kommer att analyseras och granskas i denna rapport.

Nya perspektiv och lösningar på problemet med trådorganisationen och schemaläggning av warps kan bidra till att fler uppmuntras till att använda GPU-programmering. Syftet med denna rapport är att undersöka effekterna av olika trådorganisationer i rumsliga föreningar. Förhållandet mellan objekten inom datamängder undersöks genom att beräkna antalet kollisioner som ihopslagna datamängder förorsakar. Detta görs för att förstå hur olika metoder kan påverka effektivitet och prestanda. Prestandamätningar av olika metoder inom trådorganisationer undersöks och analyseras för att hitta den mest tidseffektiva lösningen.

I denna rapport visualiseras också det erhållna resultatet av olika trådtekniker som används för att optimera beräkningshastigheterna för rumsliga föreningar. Rapporten undersöker en CPU-algoritm och två GPU-algoritmer. GPU tiderna jämförs hela tiden med exekveringstiderna på CPU:n, och GPU-implementeringarna verifieras genom att jämföra antalet kollisioner från både CPU:n och GPU:n.

Under analysdelen av rapporten jämförs och diskuteras olika implementationer med varandra. Det visade sig att skillnaden mellan en algoritm som implementerar trådtekniker och en icke-optimerad version är cirka 80 % till förmån för algoritmen som implementerar trådtekniker. Det visade sig också att den är runt 56 gånger snabbare än de rumsliga föreningarna på CPU:n.



## Nyckelord

GPU-acceleration,rumsliga föreningar, optimering av trådar, warp schedulering

# Contents

# Chapter 1

# Introduction

Spatial join is an operation where two or more datasets are combined with respect to their spatial relationship. By joining the datasets, we can discover the relationships between them and thereby make some conclusions about the properties that their collision produce. The term $spatial$ refers to an object in dimension 2 or higher while term $join$ indicates a potential intersection between the objects in the space. Spatial joins have found its purpose in the spatial databases where the objects are stored and queried in an optimized way. The reason for that lies in the fact that the database applications required spatial data types to be able to store the spatial objects [1]. The actual join consists of merging different tables from the database to discover the same or different properties that those table have in relation to each other. Apart from the databases, more complex use case like cartography emphasizes the importance of the spatial joins being implemented, for example, in the Geographical Information Systems (GIS). In this project the collisions between the rectangles from two datasets presenting the areas on the map of California are examined. That is the reason for implementing the spatial joins as they provide a way of detecting a potential object collision.

The ever-increasing performance in parallel computing where many computations are processed simultaneously has found its purpose and usage in spatial joins as well. The reason for that lies in the fact that there is no data dependency between the collision computations. Data dependency implies that one instruction is dependable on the results from the previous instruction. Thus, one object can be tested for the collision with several other objects at the same time. Consequently, the computations for collision detection can be performed in parallel without compromising the correctness of those computations.

Depending on the type of the graphics card inside a system, different programming models may be utilized. The first model, OpenCL [1] provides the user with an interface that gives the possibility to program the heterogeneous programs on various GPU:s. It is a cross-platform and enables a parallel programming of diverse processors found inside the

[1] https://www.khronos.org/opencl/

computers, servers, mobile devices and embedded platforms.

On the other side, CUDA [3] programming model is specifically meant for the NVIDIA's GPU:s. Considering that the NVIDIA graphics card (specifications about the GPU in Appendix A) is used for the spatial joins in this project, the CUDA interface is utilized throughout this project, more about it in the following chapter.

One of the important factors that may affect the performance of a GPU computation is the utilization of underlying hardware inside the GPU. The optimization of it can be achieved by organizing the threads that are performing the collision computations. As threads are the direct execution units there is a need of organizing them in an optimized way that enhances the appropriate hardware utilization.

Organizing the threads with CUDA is possible by configuring the grids of blocks and blocks of threads. Thread management can be expressed in multidimensional way, where maximum is three-dimensional presentation i.e. 3-dimensional grids with 3-dimensional blocks containing threads.

## 1.1 Background

Data collisions have found its usage and applicability in many science areas and particularly in GIS or in simulation sciences where scientists work with spatial models [2]. Also, the gaming industry implements spatial joins methods to test if objects collide in games. Due to the massive parallelism potential that GPU: s offer, a need for a user-friendly interface emerged in order to easier exploit that potential.

One way of implementing the software in order to utilize the GPU is by using the CUDA platform [3]. CUDA provides an interface for the programmer to control and process the data on the graphics card. Platform itself is an extension of C/C++ and thereby it is understandable by a majority of people in the software development.

At the hardware level the division of parallel computations is constrained by the number of processors that exist inside a system. Single core CPU: s today have reached its limits in amount of computational power they can achieve. Due to that, by having several cores working together the computational speed may increase. Thereby, GPU: s with their large number of cores are certainly a suitable solution. Even if there are multi core CPU: s today difference between CPU and GPU is significant. The number of cores on a GPU is greater and they are more suitable for data-parallel tasks where the focus lies on throughput of parallel programs [4]. On the other side CPU: s are better when it comes to complex control logic when there is need to optimize the execution of sequential programs.

Since GPU: s today have a multi core architecture it is possible to deploy a SIMT execution model on the GPU: s. SIMT is a combination of different parallelism types, namely, multiple instruction-multiple data(MIMD), single instruction-multiple data (SIMD), multi-threading

and instruction-level parallelism [4]. As data dependencies are not presented in spatial joins, using SIMT is a technique that can increase the performance dramatically.

In order to achieve the parallelism on a hardware level(GPU) Streaming Multiprocessors [4] are used. There are numerous SM inside a GPU and each SM is designed to support the concurrent execution of threads. Threads are organized into warps that are fixed size batches inside which threads are grouped and execute the same code [5]. Those warps are later scheduled on numerous SM, where SM may execute more warps at the same time.

## 1.2 Terminology

**ALU** Arithmetic Logic Unit is a combinational digital circuit performing arithmetic bitwise operations on integers in binary form

**CPU** Central Processing Unit

**CUDA** Compute Unified Device Architecture

**Data dependency** An instruction computes the result from preceding instruction

**Data rectangle** A rectangle object from $data\,rea02$ dataset

**GEO** Geography

**GIS** Geographical Information Systems

**GPGPU** General-purpose computing on GPU: s

**GPU** Graphics Processing Unit

**MIMD** Multiple instruction, multiple data

**Query rectangle** A rectangle object from $query\,rea02$ dataset

**SIMD** Single instruction, multiple data

**SIMT** Single instruction, multiple threads

**SM** Streaming Multiprocessors

**Heterogeneous program** A program is being executed on both CPU and GPU

## 1.3 Problem

One of the main concerns when dealing with hardware is to maximize its utilization while it processes data. As today's GPU: s have many cores, there is a need of managing those

cores by implementing optimized software solutions. Due to the absence of data dependencies [4] in object collisions the massive parallelism may be deployed when testing the intersection of objects.

Different thread management approaches may affect spatial joins performance in diverse ways. First, there is a need to define which dimension to use, 1D, 2D or 3D. After the dimension is chosen, the organization of threads among different blocks occurs together with managing the blocks inside grids.

As warps carry a fixed number of threads on each SM [4], thread organization across the whole GPU depends on how efficient the warp scheduling mechanisms are. In that way SIMT technique may be tested to see how well or not it correspond to different configurations of warp scheduling.

The problems above may be summarized in one question that gives the direction for the study, namely: ”How can the utilization of underlying hardware be optimized for spatial join operations using different techniques for organizing the threads inside a GPU? ”

## 1.4 Purpose

The purpose of this thesis is to give an insight about how underlying hardware can be used differently with diverse performance by comparing, analysing and discussing different approaches for the GPU solutions. Here, different organization of threads are examined on spatial joins where two datasets are merged together to find the number of collided objects. Thereby, having the number collisions as a relative thing to compare different approaches on, examination and analysis are more relevant as spatial joins are applicable in many science fields today.

## 1.5 Goal

The goals with this project are:

- Implementation of the spatial joins on the GPU.
- Implementation of the spatial joins on the CPU in order to compare it with the GPU performances.
- Optimizing the thread utilization.
- Analysis of the performance.

Thereby, this project is focused on understanding the importance and the reasons of the GPU-architecture and different thread techniques while programming on the GPU to op-

timize the computing performance, in this case spatial joins. Increasing the awareness of the eventual optimizations due to the understanding of the GPU-hardware may raise the level of interest among people in CUDA-programming. As there are numerous scientific fields where GPU-programming can be applied, by attracting more people with it can cause new use cases of the GPU-programming to emerge in different areas of science and industry.

## 1.6 Benefits, Ethics and sustainability

This project is about optimizing the underlying hardware and in order to increase the spatial joins performances, the sustainability aspect may also be included. Reason for that lies in the fact that by optimizing the utilization of hardware the overall energy consumption may be reduced and the contribution to the sustainability can occur. Assuming that the GPU takes more power for the computation then the CPU it is also performing the spatial joins much faster than the CPU and that ratio between those two components may favour the GPU in less energy consumption.

This thesis project includes also the ethical aspects that are considered during its implementation. The relevant ethical aspects that are part of the IEEE Code of the Ethics are [6]:

- to accept responsibility in making decisions consistent with the safety, health, and welfare of the public, and to disclose promptly factors that might endanger the public or the environment.
  None part of this project is a threat to safety, health or welfare of people neither to the environment.

- To be honest and realistic in stating claims or estimates based on available data.
  All of the data that is used and results that are gathered are based on the real evaluations and analyzes.

- To improve the understanding of technology; its appropriate application, and potential consequences.
  One of the main goals with this project is to increase the awareness of optimized hardware utilization in spatial joins.

- To maintain and improve our technical competence and to undertake technological tasks for others only if qualified by training or experience, or after full disclosure of pertinent limitations.
  The continuing literature study increases the area of knowledge in this type of technology which increases the relevancy of this project.

- To seek, accept, and offer honest criticism of technical work, to acknowledge and correct errors, and to credit properly the contributions of others.

This project is about analyzing and making the conclusions based on the observations that can either be positive or negative while relying on the facts and objective reasoning that are used to analyze the gathered results.

Rising the awareness of the importance of thread organization may benefit the people because of the new viewing angles toward GPGPU and can benefit the environment as energy consumption may decrease due to the less power consumption.

## 1.7 Methodology

Defining the most appropriate methodology is vital to a successful project. The same applies for this thesis. Implementing methods that create relevant and reliable outcomes increases the chance of project being valid and accepted by others. In this project following methods are used:

- **Research** In order to have appropriate knowledge in the area the literature study is conducted. Fields that are investigated are: CUDA programming, GPU-architecture and thread organization. A successful research needs the right research methods,approach and strategy that are further explained in the Chapter 3.
- **Design** After the CUDA environment is familiar and theory behind GPU architecture is known the design part of project may start. This method includes the definition of the important aspects for the GPU-programming. Data types, data quantity and quality aspects are considered. Knowing them can help with design of the software solutions. When the designs are defined and modeled, the implementation may start.
- **Testing** Different approaches are tested in order to generate the resulting data.
- **Data Collection** By knowing which data is relevant in this project the correct analysis of the work may be conducted. Types of data, methods of extracting the particular data and reasons why they are applicable are discussed.
- **Methods For Analysis** After the correct data have been gathered it has to be analyzed in order to make the conclusions about the more efficient solutions to the problem.

## 1.8 Delimitation

There are software delimitations in form of implementation of the brute-force broad-phase spatial join algorithm [7]. Better software solutions for the spatial joins can be examined in

the future studies. The most important part of this thesis is to realize the impact of threads that can be differently organized and scheduled on the GPU.

## 1.9 Outline

After the introduction, there are five more chapters to come. Firstly, the theoretic background about the field of GPU is examined and analysed. Then, in the third chapter the methodology defines how the theory can be applied to the project. Chapter 4 describes the implementation and presentation of the datasets that are used in this project. After Chapter 4, the experiment is described in chapter 5. Chapter 6 visualizes and analyses the gathered results and finally in Chapter 7 the overall conclusion about the conducted and future work is drawn.

# Chapter 2

# Background

This chapter introduces a reader with spatial joins and its types. Then, CUDA environment, implementation methods and differences between C and CUDA code are presented. Summarizing with an architectural overview of the GPU, describing threads, streaming multiprocessors and warps.

## 2.1 Spatial joins

Spatial join is a fundamental operation for many spatial queries in GIS [8]. Knowing what are the nearest places to a location may be interesting if, for example, a person wants to know what places of interest are close to her/him. The spatial join finds pairs of objects from two sets of spatial data that satisfy a condition involving spatial attribute values, such as overlap [9].

Rather than just looking at one layer of the map a more interesting feature is to be able to add a layer of points of the interest. Then by intersecting those two maps it could be possible to find the relationship between those two layers. The result of joining two layers would be that a person can find near points of interest (e.g restaurant, coffee shop, hotel) that are close to her/him. This types of problems can be solved using spatial joins algorithms.

There are two types of datasets that may be analysed and used in spatial joins, namely those that have indexed data and those that have unindexed data. According to [9] the spatial join is generally fastest if the data sets are indexed. Thereby, having two layers of maps that are both indexed inside their datasets is preferable to use in process of spatial joins. There are two approaches to the spatial joins [7]:

- **Broad phase** collision tests are conservative - usually based on bounding volumes

only - however fast in order to quickly prune away pairs of objects that do not collide with each other.
- **Narrow phase** collision tests usually compute exact contact points and normal and thus are much more costly, however performed only for the pairs of the potentially colliding set. Such an approach is very suitable when a lot of the objects are not colliding. Thereby, the appropriate implementation for the objects with the unshaped volume would be to first use the broad phase algorithm and prune away the non-colliding objects and then compute the exact colliding points using narrow phase for the remaining objects.

In this thesis project the broad phase approach is implemented as the interest is in fast detection of collided rectangles by their volume. Rectangles that will be used inside the datasets already have well shaped volume and there is no need for further accuracy by implementing narrow phase.

### 2.1.1 Broad-Phase Algorithms

The brute-force implementation of broad phase for $n$ objects has a complexity of $O(n^2)$ [7]. The implementation includes testing each object with the remaining objects in a residing space. Apart from the brute force algorithm, there are alternative algorithms, like: sort and sweep and spatial subdivision and they achieve an average complexity of $O(n \log n)$ (their worst-case complexity remains $O(n^2)$) [7]. The following text explains the details of those two algorithms.

#### Sort and Sweep

To limit the number of objects that should be tested to see if they collide(intersect) with each other, sort and sweep algorithm may come to use. That is possible to achieve by sorting start (lower bound) and end points(upper bound) of the object on an axis. Those rectangles having ranges that overlap are further tested to see if they collide. Objects whose collision ranges do not interfere in one of the current dimensions cannot possibly collide with each other. For objects to collide they need to intersect in all the dimensions of the space they are residing in.

According to Figure 2.1 two-dimensional rectangles are first tested by looking at their ranges on the x-axis. First, the list containing $2n$ entries is used to store all of the points (both start and end points) of each object. Then, list is sorted in ascending order so that later when traversing it, start points ($S$ in Figure 2.1) could be found and the current object added to the list of active objects. The active objects are those object whose interval intersect with the other object's interval. Whenever an end point ($E$ in Figure 2.1) is found, corresponding object is removed from the list of active objects. Thereby some unnecessary testing is avoided by diminishing the number of tested objects.

Collision tests for an object $A$ are performed only between that object and all other objects that are currently in the list of active objects when object $A$ itself is added to this list of active objects. In the example of Figure 2.1 it is possible to see that all of the rectangles $Ri(i = 1, 2, 3)$ are intersecting with another object. Thereby, all of them will be examined to see if they do collide in the higher dimensions.

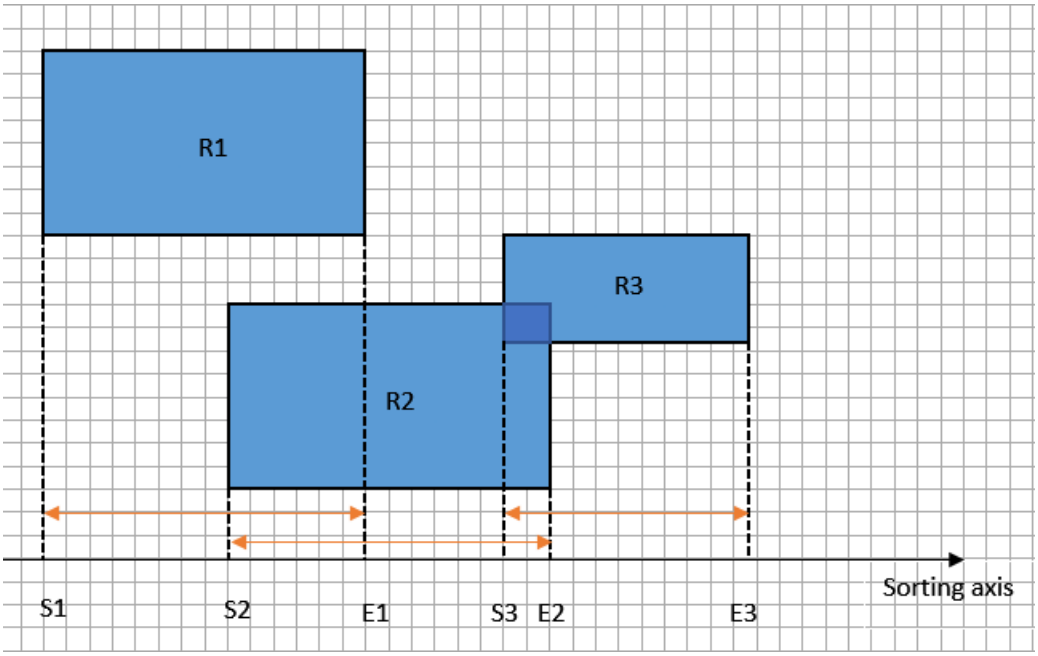


Figure 2.1: Sort and Sweep visualization

### Spatial Subdivision

Instead of using the sweep and sort algorithm, the spatial subdivision approach may be implemented. Spatial subdivision divides the space into a grid, such that each cell (all of them are the same size) is at least as large as the largest object in the space. Every cell has a list of objects whose parts are within a cell. Two objects that are belonging to the same cell or two neighbouring cells are tested to see if they intersect with each other. According to the Figure 2.2 following object pairs would be tested to see if they interact with each other:

- $R1$ with $R2$ and $R3$
- $R2$ with $R3$, $R1$ and $R4$
- $R3$ with $R2$ and $R1$
- $R4$ with $R2$

Apparently, the problems that may occur are unnecessary computations that would be performed. Namely, same pairs would be tested multiple times. An alternative way that

would be to able to reduce the problem is to assign to each cell the list of all objects whose bounding volume intersects the cell[7]. In this case, an object can appear in up to $2^d$ cells, where $d$ is the dimension of the spatial subdivision. Only the objects that appear in the same cell and at least one of them have the midpoint in that cell are tested to see if they intersect with each other. The midpoint of the rectangle is the intersecting point of the diagonals of the rectangle. The simplest implementation of spatial subdivision creates a list of object IDs along with a hashing of the cell IDs in which they reside, sorts this list by cell ID, and then traverses swaths of identical cell IDs, performing collision tests between all objects that share the same cell ID [7]. Looking again at the Figure 2.1 the following pair would be tested:

- $R1$ with $R3$
- $R3$ with $R1$

In this approach the number of examined collisions is reduced from previously 8 to 2. However, even by using with this approach, there are redundant computations. Rectangle pair $R1$ and $R3$ is tested two times which is unnecessary. To with this problem a more sophisticated solution is proposed in form of hierarchical grids [7] which is beyond the scope of this project.

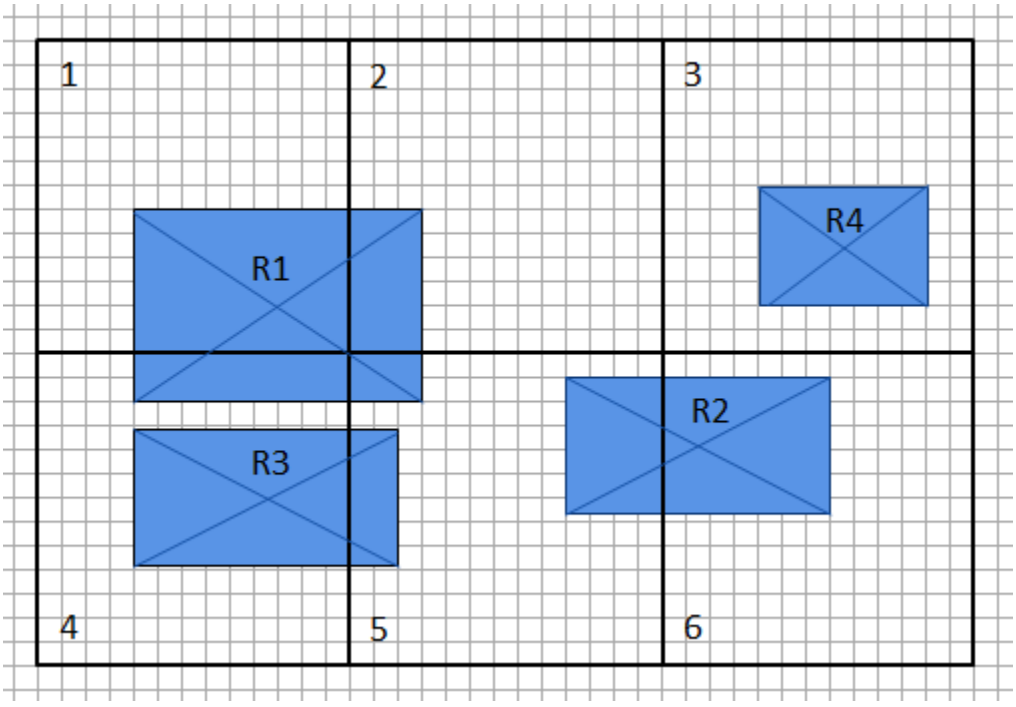


Figure 2.2: En example of 2D Spatial Subdivision of rectangle objects

## 2.2 CUDA environment

Since current GPUs have more than thousand cores located inside them, the need for the GPU programmability is emphasized because of their parallelism potential. Programming the GPU has its use cases in many science fields and the gaming industry thereby it is crucial to have an optimized and intuitive language that will deploy the power of graphics card in various kinds of science researches. By using the CUDA programing language, developers may exploit the parallelism by writing C/C++ similar code that will run on thousands of threads [10]. The CUDA programming model enables the developers to execute applications on heterogeneous computing that consists of CPUs combined with GPUs, each with its own memory separated by a *PCI-Express bus*[1]. Starting from CUDA 6, NVIDIA introduced an improved programming model called *Unified Memory*. Unified Memory (Figure 2.3) bridges the divide between CPU and GPU memory spaces by having a common virtual address space and this improvement allows the developer to access both the CPU and GPU memory using a single pointer, while the system automatically migrates the data between them [4]. The figure shows that the Kepler architecture utilizes the Unified Memory however that concerns Maxwell architecture, used in this project, as well. The reason for that is because Maxwell is the follower of the Kepler architecture that was the pioneer in the Unified Memory [11]. The CUDA system software automatically migrates data allocated in Unified Memory between GPU and CPU, so that it looks like CPU memory to code running on the CPU, and like GPU memory to code running on the GPU. In this project CUDA 8 [12] will be used which means that the Unified Memory feature is also included as this version inherits that from the previous ones. Syntax and implementation differences, for vector additions, between C and CUDA are described under for the comparison.

**C vector addition**

```
void addition(int *x, int *y,
         int *sum, int *N){
    int i;
    for (i = 0; i < N; i++) {
   sum[i] = x[i] + y[i];
  }
}
```

**CUDA vector addition**

```
__global__ void addition(int *x,
        int *y, int *sum, int *N){
        int tid = threadIdx.x;
  if(tid < N){
     sum[tid] = x[tid] + y[tid];
  }
}
```

Considering the fact that C implementation is sequential while CUDA is mainly parallel the implementation differences are inevitable. The first difference to notice is the absence of **for**-loop in CUDA code. The reason for that lies in the fact that the code is executed by

[1] *Peripheral Component Interconnect* is a local computer bus for connecting hardware devices inside a computer.

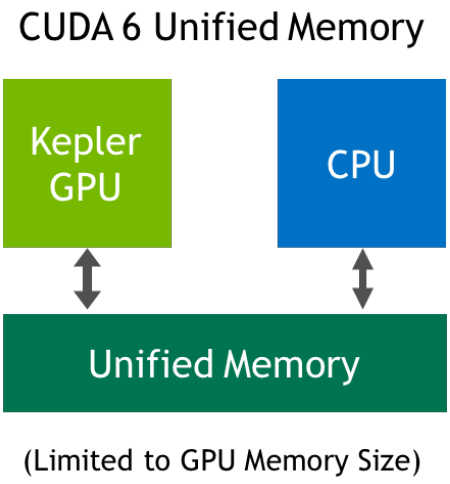


Figure 2.3: An illustration of CUDA's Unified Memory [12]

multiple threads(SIMT) and thereby, as there are built-in thread coordinate variables that replace the vector index (`threadIdx.x`), there is no need of having a **for**-loop. More about function calls and thread organizations in Section 2.3.1.

The second difference is that each function being executed on a GPU, called a kernel function, is denoted with a prefix **global** while the C language does not have a prefix of that kind, just a return type.

Besides that, CUDA implementation also includes an **if** statement to ensure that the thread indexes are not larger than the total length of the array. The reason for writing a boundary check is because when launching the kernel function, the number of threads being invoked can happen to be larger than the total amount of needed threads. To make sure that those extra threads are excluded the $if$ statement checks that the indexes, that are later used for the array, are not larger than the total number of elements in the array.

Nevertheless, there are also some similarities when it comes to memory management and memory functions. Namely, both CUDA and C have similar dynamic memory management as presented in Table 1. Firstly, the memory must be allocated with `cudaMalloc` function. When the memory has been allocated it can be instantiated by calling the `cudaMemset` function by filling the allocated memory space with values. At the end, when the data inside the allocated partition of memory is no longer needed it can be deallocated just as in C language with function `cudaFree`.

One difference although is function `cudaMemcpy`. In CUDA it is used to transfer the data between CPU and GPU:

```
cudaMemcpy (void* dst,void* src,size_t count,cudaMemcpyKind kind)
```

There are also `dst` and `src` variables defining source and destination for the data transfers while `count` defines the size of the total amount of the data to be transferred. Apart from those three parameters the last one `cudaMemcpyKind` clarifies one the four direc-

Table 2.1: Memory management functions in CUDA and C

| C functions | CUDA functions |
|---|---|
| malloc | cudaMalloc |
| memset | cudaMemset |
| memcpy | cudaMemcpy |
| free | cudaFree |

tions of data flow between $host$ (CPU) and/or $device$ (GPU) [4]:

- `cudaMemcpyHostToHost` Transfer the data between two different places on the CPU.
- `cudaMemcpyHostToDevice` Transfer the data from CPU to GPU.
- `cudaMemcpyDeviceToHost` Transfer the data from GPU to CPU.
- `cudaMemcpyDeviceToDevice` Transfer the data between two different places on the GPU.

## 2.3 GPU architectural overview

As GPU is not a standalone platform there is a need of combining it with CPU to make the usage of its potential. As stated before GPU operates in conjunction with CPU through PCI-Express bus. That is reason why CUDA applications are heterogeneous as they partially compute on CPU and GPU. Code that is executed on CPU is $hostcode$ while on the GPU side code is denoted as $devicecode$. GPU is used to enhance the code that may use the parallelism power and therefore when there are computational intensive applications there is often a potential for parallel implementation. There are 2 important metrics that describe the GPU performance [4]:

- Peak computational performance
- Memory bandwidth

Peak computational performance indicates how many single or double - precisions floating points calculations may be computed per second and the measurement is `gflops` (billions of floating-operations per second) or `tflops` (trillion of floating-operations per second). Memory bandwidth measures the speed at which the data can be read from or written to the memory and is denoted in GB/s (Gigabytes per second).

The specifications about the working environment and graphics card used in this project are presented in Appendix A.

### 2.3.1 Threads

Launching the kernel function requires the threads to be organized in order to compute the data on the GPU. When the kernel function is launched, the code is executed on the GPU. In order to call a kernel function one must specify the dimension and configuration for both grid and thread blocks.

```
//continuing from the previous example
int main(int argc, char **argv){
 .
 .
 int num = 1000; // number of elements
 dim3 block (10); // number of threads in block per dimension
 // counting the number of thread blocks per each dimension
 dim3 grid ((num + block.x - 1)/ block.x);
 addition <<<grid, block >>> (x, y, sum, N);
 .
 .
}
```

The `dim3` data type is used by both grid and blocks. It has three unsigned integer (`dim (x, y, z)`) fields and unused fields are initialized to 1 and ignored [4].

All the threads that are invoked are executing the same instructions inside the kernel function that specifies them. Threads that are executed are organized into a three-level hierarchy [13]. At the top, there is a $grid$ that can either be one or two-dimensional consisting of thread blocks where each block has unique coordinates. Blocks can be up to three-dimensional arrays of threads with distinct coordinates inside blocks. An example with two-dimensional grid with two-dimensional thread blocks is given in Figure 2.4 where coordinates are defined as follows:

- `(blockIdx.x, blockIdx.y)` specifies the coordinates for each thread block inside the grid.
- `(threadIdx.x, threadIdx.y)` specifies the coordinates for each thread inside the block.

For instance, in the above vector addition example the coordinates for thread blocks are `blocksIdx.x` and for threads are `threadIdx.x` because the implementation consisted of single dimensional grids and thread blocks.

The grid formula `dim3 grid (((num + block.x - 1)/ block.x))` is defined in that manner so that the total amount of threads that are later executing the functions can

be rounded up. One of the reasons for not just dividing the total number of elements with the number of thread blocks is following: Let's say that the total number of elements is 7 which when divided with the number of thread blocks, for example 2, will be rounded down to 3 and that number is the count of threads inside those two blocks. By having that, the kernel function will be called as follows: `addition<<<2, 3>>> (x, y, sum, 7)`. Only 6 threads for 7 elements long vector is not enough and thereby the error will occur. That is the reason why it is important to round up the number of thread blocks inside the `dim3 grid` even if the total number of threads may be larger than the total number of elements. Controlling the number of threads inside the kernel function is a solution to that problem, as it is written in the vector addition example `if (ix < N)` where `ix = threadIdx.x`.

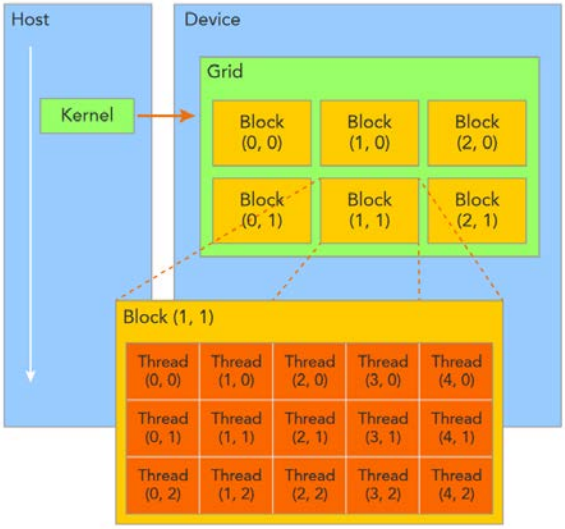


Figure 2.4: Two-dimensional grid consisting of two-dimensional thread blocks [4]

### 2.3.2 Streaming Multiprocessors

The GPU that is used in this project has a Maxwell architecture that consists of 5 SMMs and each of them has 128 CUDA cores (Appendix B). As the SMs inside the GPU support concurrent execution of numerous threads by having several of them on a GPU huge number of threads may be executed simultaneously. When a function is launched on a GPU (kernel function) the thread blocks inside a grid are arranged among free SMs. Threads inside the thread block that is assigned to a specific SM can execute concurrently only on that specific SM. However, multiple blocks may be attached to one SM at the same time and then they are scheduled depending on the availability of that SM's resources. SM divide the thread blocks attached to it into warps consisting of fixed number of threads that it then schedules.

The resources of SM are memory and registers. The memory is divided among the thread blocks that are inside the SM and registers are intended for the threads. Even if the warps

are scheduled on a specific SM it doesn't mean that they are active. A warp that idles causes the SM to schedule another available warp from any thread block that is inside the same SM. Switching between concurrent warps has no overhead because hardware resources are partitioned among all threads and blocks on an SM, so the state of the newly scheduled warp is already stored on the SM [4].

### 2.3.3 Warps

The main execution unit of SMs are warps. When deploying the thread blocks inside the grid threads are distributed among the SMs. Later, the threads inside the blocks are divided into the warps, containing 32 threads each. All the threads inside a warp execute the same instruction (SIMT fashion) while each thread carries out the operation on its own private data [4]. The difference between the hardware and software view is described in Figure 2.5. There it is noticeable that threads are organized into the fixed size warps on the hardware level, while the on the software level the threads are managed inside the thread blocks. Thereby, even if the threads are software configured to be in a one-, two- or three-dimensional space, on the hardware level they are organized in one dimension. When it comes to thread indexing, every thread has a distinct thread ID that is stored inside a CUDA build-in variable depending on a dimension that the thread is arranged in, as discussed in section 2.3.1.

To access a thread the dimension inside which it resides can be converted into the one dimension using the $x$ as the first, $y$ as the second and $z$ as the final dimension. Accessing thread requires the thread block (up to three dimension) to be specified as well. Here is the list of possible combinations for accessing the threads, depending on different dimensional configurations:

- One-dimensional thread block: `threadIdx.x`
- Two-dimensional thread block: `threadIdx.y * blockDim.x + threadIdx.x`
- Three-dimensional thread block: `threadIdx.z * blockDim.y * blockDim.x + threadIdx.y * blockDim.x + threadIdx.x`

Therefore, in order to specify a particular thread, blocks coordinates should be considered as well.

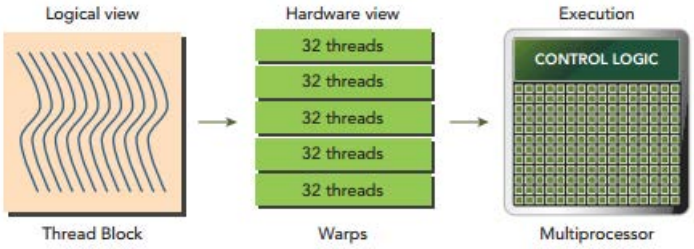


Figure 2.5: En illustration of hardware and software view on threads [4]

# Chapter 3

# Methodology

In this chapter the research, analysis and data collection methods are considered in order to steer the direction of the ongoing work.

## 3.1 Methodologies and the implemented methods

There are several categories for the methodology that are implemented in this bachelor thesis project. The following top-down range looks as follows [14]:

- **Quantitative research method**
- **Philosophical assumption** - Realism
- **Research Methods** - Experimental
- **Research Approach** - Inductive
- **Research Strategy** / **Design (Methodology)** - Experimental
- **Data Collection Method** - Experiment
- **Data Analysis Method** - Statistics
- **Quality Assurance** - Methods used to assure the quality are: Validity, Reliability, Replicability and Ethics
- **Contributions and results** - Presentation of the whole research.

$Quantitative$ research method includes the experimental measurements that may prove or confute the theories and statements or as in this project the functionalities of computer systems and its parts. For the $quantitative$ project to be valid, large number of datasets should be included to have more reliable results. As this bachelor thesis project is about

proving the importance of thread organization on the spatial joins with large datasets used for benchmarking, it has $quantitative$ characteristics. Philosophical assumption implemented in this research is $Realism$ because that approach implies the observation of the phenomenon in order to gather the reliable and valid data that can lately be reapplied and reused. When it comes to further research methods that are meant for the starting, executing and finalizing the research, $experimental$ research method is utilized. It studies the different variable that may be controlled by the observant and the results from different inputs are later analysed and discussed which is the case in this project as performances are investigated.

For drawing the conclusions and establishing what is true or false, different research approaches can be implemented [14]. This report that has a quantitative structure takes the facts and combine them in order to make the conclusions about the measurements that are carried out. The large data sets are combined to see the relationships between different approaches. Later, results are gathered based on the collected measurements. When the results are gathered, the question from the start concerning the possibility of the thread optimization may be answered which implies that the $deductive$ approach is used in this bachelor project.

The research strategy guidelines the whole research by organizing, planning designing and conducting the research [14].In this project the $experimental$ design is used. The experimental research strategy controls the circumstances that may affect the results. Different variables that affect the performances are tested and the reasons for the different results are discussed. As this project includes a large data sets from which the data is collected the $experiment$ data collection method is implemented.

After collecting data, an important aspect to consider is the appropriate data analysis so that the correct conclusions may be drawn. Measuring the performance includes a current result related to some other observation. The idea is to have some other object to relate the measurements to so that the potential improvements or decreases in the performance can be noticed.The most descriptive method to use in this project would be to utilize the $statistics$ for the data analysis.

To ensure the quality of the quantitative research material with deductive approach the following methods must be implemented [14]:

- $Validity$ method certifies that the measurements are those that are expected to be.
- $Reliability$ method is responsible for the consistency of the results for each test.
- $Replicability$ method ensures that someone else can reapply the methods with the same variables and get the same results which requires well described procedures.
- $Ethics$ represents all the moral principles in the whole process of research project. Respective methods take care of the protection of participants, their privacy, avoiding to put the pressure on them so that they can decide on their own what they are going to write about while threating material with confidentiality.

The last part of the project consists of the contribution and results that are being described.

The description covers the theory and research methods (in the beginning), the method applications and the reasons for the implementation of specific methods on the research project, the collected data from the observations and finally the analysis and discussion about the results.

## 3.2 Implementation methods and models

This project is about spatial joins between the objects from two different datasets. Spatial joins include data collisions and thereby a method for collision detection is needed. A proper model potentiates a fact that later implementation in the code can be reliable and satisfactory. The first thing to consider, about the data collisions, is that all the different scenarios should be included. As a collision between the objects is the intersection of their parts, all the cases where they overlap each other need to be contained in the model for detection.

The objects being examined in this project are parallel rectangles in two-dimensional space that have two coordinates that are used to detect if the objects collide with each other.

By looking at the *Cartesian*[1] values for both $x$ and $y$ in the two-dimensional coordinate space of the corresponding objects, the collision can be detected. The calculation of $x$ and $y$ ranges from respective objects may reveal the eventual overlap of both the $x$ and $y$ ranges that would indicate a collision. Important to notice is that it is not sufficient that only one, either $x$ or $y$, range intersects as both need to overlap for the collision to happen. One example of the similar solution is presented in Figure 3.1. According to the image, the only objects that collide are $R4$ and $R3$ as both of their ranges intersect which is not the case with $R1$ and $R2$ as their $y$ ranges are not overlapping.

In case the $x$ ranges of two rectangles intersect that implies that one or both $x$ coordinates of one rectangle are between the high and low $x$ values of other one. On the other hand, if $y$ ranges intersect that means low and/or high $y$ coordinate of one rectangle is in the $y$ range of another. Thereby, a summary of conditions needed to be satisfied for the rectangles $A$ and $B$ to intersect with each other is as follows, described with equivalence laws [2]:

$$(Ax_{low} \leq Bx_{high}) \wedge (Ay_{low} \leq By_{high}) \wedge (Ax_{high} \geq Bx_{low}) \wedge (Ay_{high} \geq By_{low})$$

The reason for having an equal sign in the condition statements is because even if the objects touch each other they are colliding and those cases should be included as well.

[1] *Cartesian coordinate system* defines each point in a plane uniquely by pair of numerical signed values that present the distances between the $Origo$(point where axis intersect) and the point on the appropriate axis that the orthogonal projection of respective point produce.

[2] http://www.cs.put.poznan.pl/ksiek/latexmath.html

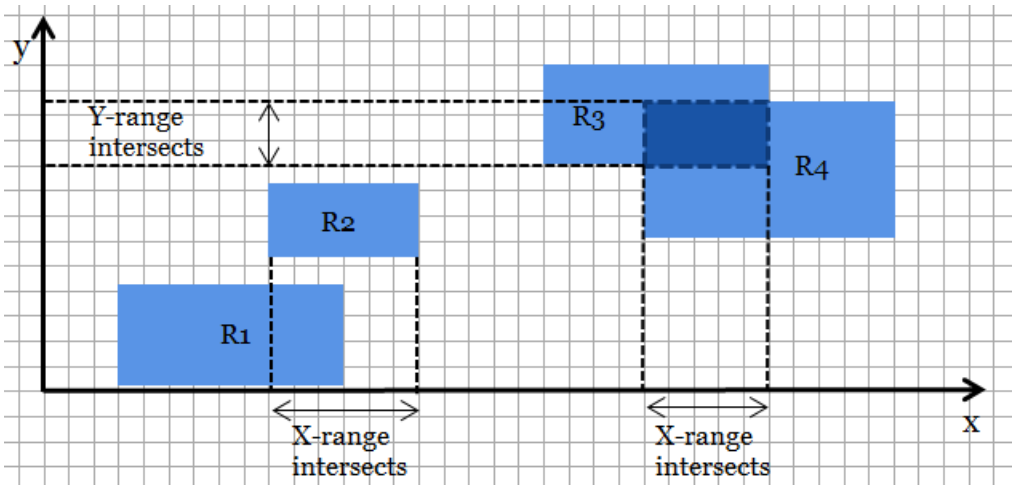


Figure 3.1: Colliding and non-colliding objects

Proving that the above conditions need to be true for the collision to happen can be done in two ways either by presenting all the possible cases or using the contradiction. Knowing the fact that if one of the ranges, of one rectangle, does not intersect with the another one's range than it is possible to conclude that the rectangles do not collide as presented in Figure 3.1 with rectangles $R1$ and $R2$. Thereby, if one of the conditions is not true there is no intersection which can be presented as a following condition:

$$(Ax_{low} > Bx_{high}) \vee (Ay_{low} > By_{high}) \vee (Ax_{high} < Bx_{low}) \vee (Ay_{high} < By_{low})$$

The negation of that statement would mean that the rectangles collide and that is written as follows:

$$\neg((Ax_{low} > Bx_{high}) \vee (Ay_{low} > By_{high}) \vee (Ax_{high} < Bx_{low}) \vee (Ay_{high} < By_{low}))$$

Which can be further rewritten using De Morgan's laws [15] in the following condition:

$$\begin{gathered}\neg(Ax_{low} > Bx_{high}) \wedge \neg(Ay_{low} > By_{high}) \wedge \neg(Ax_{high} < Bx_{low}) \wedge \neg(Ay_{high} < By_{low}) \\ \equiv \\ (Ax_{low} \leq Bx_{high}) \wedge (Ay_{low} \leq By_{high}) \wedge (Ax_{high} \geq Bx_{low}) \wedge (Ay_{high} \geq By_{low})\end{gathered}$$

Thereby the condition for the collision detection is proved as a negation of a non-collision condition.

A model representing the computational process is given in Figure 3.2. In the beginning rectangles are read from two different datasets and stored inside two array data structures so that they can be accessed faster. Inside the loop $R_i$ rectangle is obtained from $S_1$ to see if it collides with the rectangles from $S_2$ dataset. In order to test it, $R_i$ rectangle is compared with $R_j$ rectangle from the $S_2$ dataset where $R_j$ and $R_i$ rectangles are indexed objects in two different arrays(two datasets). The intersection between $R_i$ and $R_j$ rectangles is tested using the previously defined conditions as illustrated in Figure 3.3. The collision counter is incremented each time a rectangle $R_i$ collides with $R_j$ while $j$ variable is incremented after every test in order to properly index all of the rectangles from the $S_2$ dataset. When

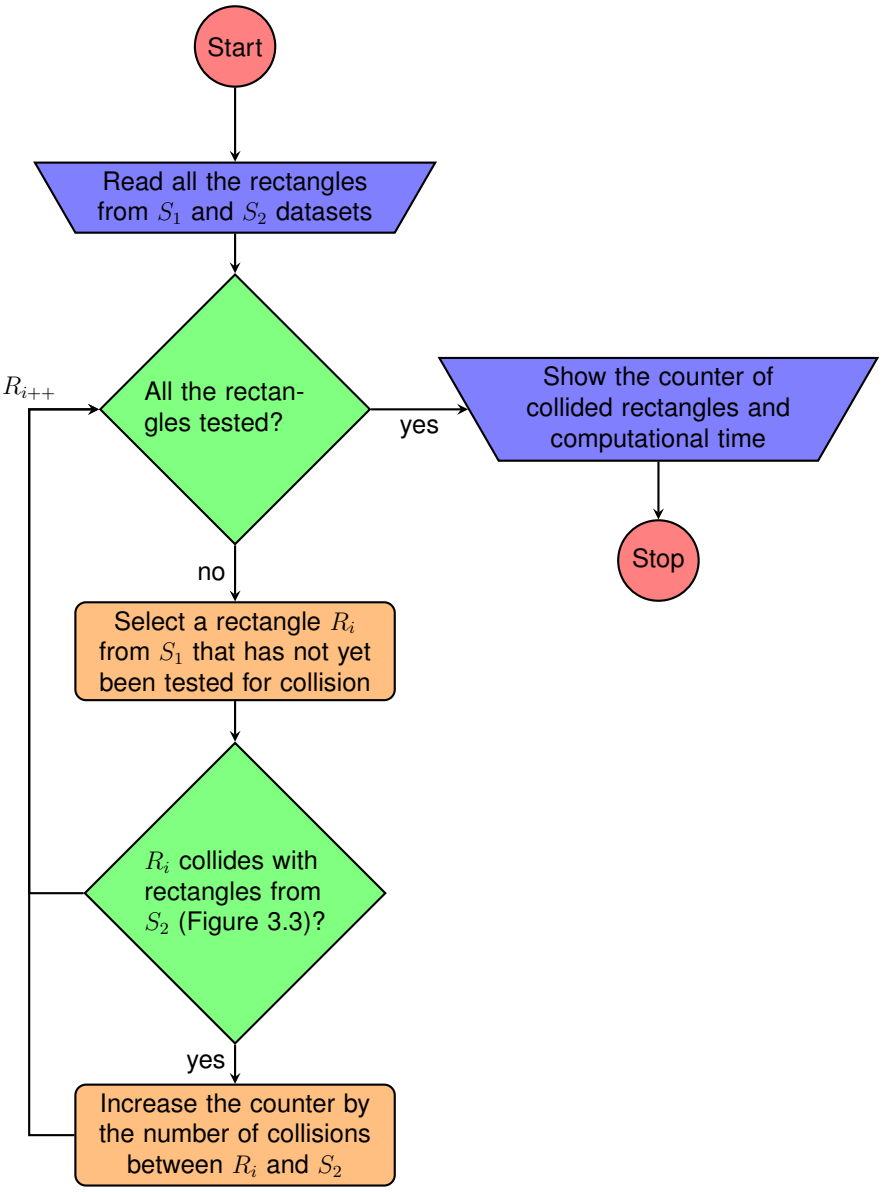


Figure 3.2: Flowchart describing the process of computation

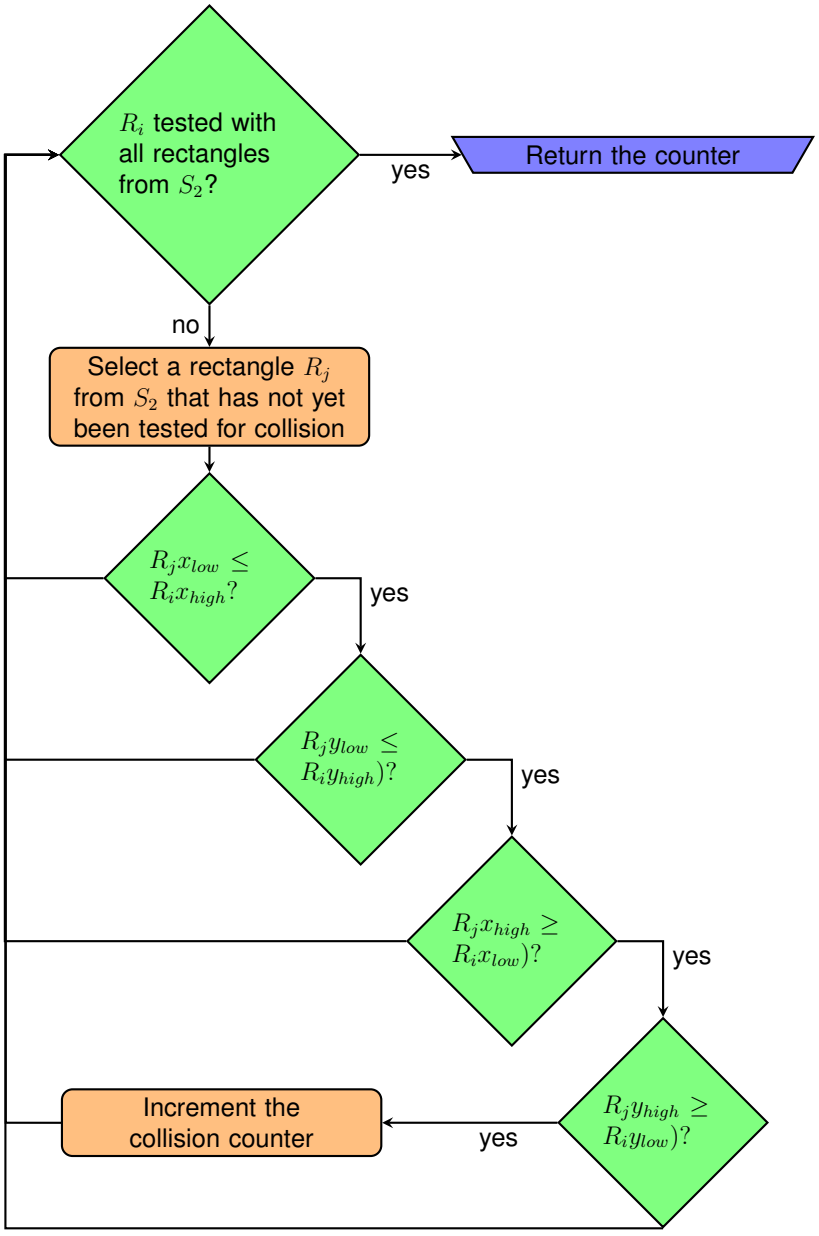


Figure 3.3: Collision detection process

the $R_i$ is tested with every rectangle from $S_2$ dataset the $i$ variable may be incremented for the same reasons as before so that the next rectangle from $S_1$ can be evaluated.

To measure the time, after each execution the proper approach should be taken so that the results can be relevant. The GPU execution times are compared with the time it takes to compute the spatial join process on CPU. That means, the algorithms are executed on both CPU and GPU and those results are then compared. Measuring the time for execution concerns three factors that may affect it:

- CPU or GPU, where is the code executed?
- Which GPU algorithm is implemented?
- Thread organization, how are they organized?

After all the rectangles have been evaluated, the final collision counter is shown together with the time it took to compute the data on both GPU and CPU.

## 3.3 Data Collection

The research strategy that is applied in this report is quantitative which means that different observations are used to make the conclusions about the spatial joins on the GPU with different thread organizations.

The type of the data that is collected in this report consists of different time measurements. The time it takes for the GPU to compute the data is compared with CPU in order to confirm the eventual time improvement or to show the decrease in performance. For the measurements to be credible it is important to measure also the time it takes to transfer the data between the host(CPU) and device(GPU). Problems that may cause the data to be wrong:

- CPU and/or GPU compiler optimizations may cause the time to be different in same several identical executions. That is the reason why the same benchmarks are executed several times to find average values.
- Potential overheads with time functions because the CPU is measuring the time and not GPU.
- Quantity errors in form of insufficient test may cause the results to be unreliable.
- Omission errors may results in eventual mistakes when noting the data can give the invalid results.
- Not all the possible cases are tested properly, in order to make a combination of $n$ cases $2^n$ tests should be considered.

For this project to be reapplied the following hardware and software is needed:

- System specs (Appendix A).
- CUDA version 8.
- Software solutions described in Chapter 4.
- NVIDIA Nsight Eclipse Edition[3] editor used for writing the programs

To collect data, executions start with low number of rectangles and then increase for each next execution until the whole datasets have been included in the tests. The reason for that is because the low-intensive computations should be included as well to see if the GPU is more efficient even then because it takes time to send data between host and device. Different thread organizations are examined as well, starting with one-dimensional continuing with two-dimensional and ending with three-dimensional thread organization. For each configuration, time is examined. After the execution is done time performance is written to a file where the information about the previous executions is stored.

## 3.4 Statistical measurements

In this experiment, there are numerous benchmarks for the same implementations in order to minimize the experimental errors. Considering that, statistical methods are needed in order to have an average value for the comparison of the implementations. Apart from having the mean value for the results, an important issue to consider is also the percentage of how much the measures disperse from the mean value which can be calculated using the formula for coefficient of variation. Calculating the coefficient of variation requires the standard deviation to be calculated as well. Thereby, formulas needed for the statistical measurements are:

- Mean value: $\overline{x} = \frac{1}{N}\sum_{i=1}^{N}(x_i)$
- Standard deviation: $\sigma = \sqrt{\frac{1}{N-1}\sum_{i=1}^{N}(x_i - \overline{x})^2}$
- Coefficient of variation: $c_v = \sigma/\overline{x}$

Where $x_i$ are different time measurements and $N$ represent the number of benchmarks.

### Results presentation

After storing and collecting data, a graphical presentation of the gathered results may be conducted. The results are presented in form of graphs in order to simplify their analysis. The data from the files is copied to the sheet where all the times are placed. Then, the graphs that are used as presentations consume the written data in order to visualize the results.

[3] https://developer.nvidia.com/nsight-eclipse-edition

## 3.5 Methods for Analysis

When the results have been gatherer and visualized, the analysis may begin. The GPU-execution time is compared to the CPU depending on the following factors:

- Where is the code executed?
- Which of the GPU algorithms is benchmarked?
- How are the thread organized?

Obtaining the possible configurations of threads is vital for the later analysis. The results are examined while considering the hardware that is used. The way in which the SMs are distributed among the GPU and the number of cores is related to the implementations and later results. Discussing the relation between the execution time and different implementations gives an insight about the overall impact of various thread configurations. The results are also compared to what has been expected considering the SMs and warps.

As stated in the beginning of the chapter the descriptive method that are used are $statistics$ because the performances are measured and compared to each other. The comparisons is based on higher performance i.e. the faster the better. CPU is correlated to the GPU while different GPU implementations are compared to each other as well. In order to verify that the implementation on the GPU is correct the number of collisions between two datasets is correlated to the CPU collision counter.

# Chapter 4

# Implementation

In this chapter a reader gets introduced with brute-force algorithm that is used in this project as it is the most upfront approach to solving a problem of spatial joins. The idea with this project is to focus on thread optimization and not software solutions. There are two algorithms described, one optimizing threads and another one not. Both algorithms are using the rectangles from two datasets to compute the number of colliding rectangles when two datasets are joined.

## 4.1 Rectangle presentation

In this project, different array data structures are used to store the data, i.e. rectangles. The dataset that contains the rectangles and that are implemented in this project, represents the map of California and is named $data\,rea02$ [16]. That map consists of around 2 million objects i.e. rectangles presenting regions and subregions of California. On the other side, there is another dataset, $query\,rea02$, that is used to make the process of spatial joins [16] by implementing data collision between it and $data\,rea02$ dataset. $Query\,rea02$ has roughly 200 000 objects and each of them should collide with one object from the map dataset. As presented in the Figure 4.1 $rea02$ datasets both represent the California however they differ in the amount of rectangles as $query\,rea02$ dataset has 10 times fewer objects then $data\,rea02$. When it comes to the rectangles inside $rea02$ datasets in the code they are stored in an array with the following struct[1] [16]:

- $x_{low}$ - $R_i[0].l$
- $x_{high}$ - $R_i[0].h$
- $y_{low}$ - $R_i[1].l$

[1] A group of variables under the same name, allowing the access to those variable using single pointer

- $y_{high}$ - $R_i[1].h$

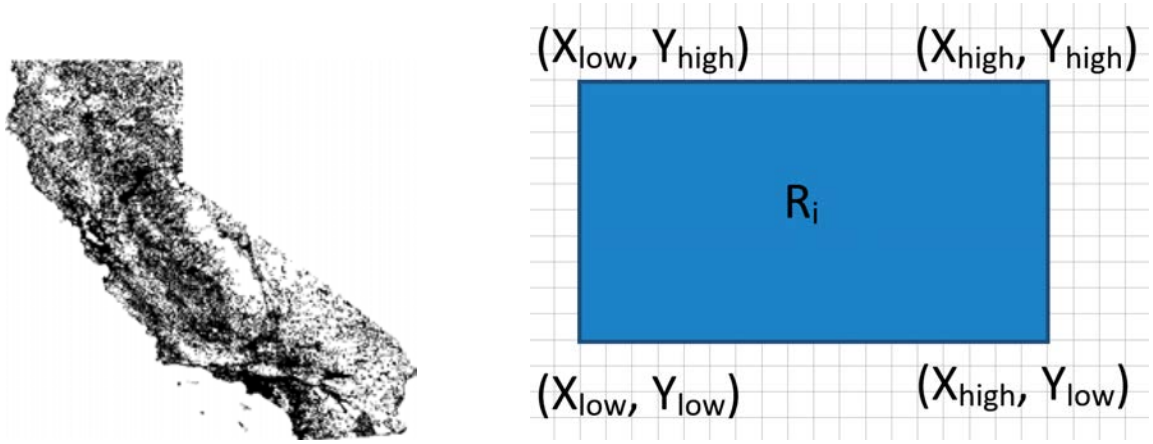


Figure 4.1: $Rea02$ rectangles (right) presenting California(left)

## 4.2 Code implementation

The starting point consisted of making the spatial join process on the CPU. As stated before, in the Chapter 3, broad phase algorithm was implemented. The algorithm is described in pseudo form named *Non-optimized* algorithm. That algorithm is for the GPU and the CPU implementation differed in the absence of function `addAtomic()` and `threadIdx.x` variable as they are used by the GPU, more about the GPU part bellow. Instead of them, the CPU code included a `counter` variable that increments on each rectangles collision from the two datasets. The CPU implementation had also one more **for** loop to iterate over the $data\,rea02$ so that for each $query$ rectangle the whole $data\,rea02$ is tested for eventual collisions. On the other hand, GPU consisted of two algorithms. The Non-optimized algorithm was implemented with `addAtomic()` as illustrated in the pseudo form. The reason for having only one **for** loop instead of two is because of the built-in variables that are used when the kernel function, in this case **atomCollisionCounter**, is executed with many threads. Each thread has its index $q_{inx}$ that represents the index of the rectangles from the $query$ array which is tested for collision with the rectangles from $data$ rectangles indexed with $d_{inx}$. Later the `addAtomic()` is used to increment the value of the $counter$. The function `addAtomic()` is utilized as it provides the insurance that other threads cannot interfere with the thread trying to update the $counter$ variable and thus the final value is correct. Lastly, the important difference between CPU and GPU implementation was the initial check for the thread index to not index out of the bounds of the $query$ array, as written in the line nr 4 of the Non-optimized algorithm.

Due to the associative and commutative properties of addition, the elements of the elements of the array can be summed in any order. That opens a new perspective to a

implementation of collision counter which can be calculated performing parallel addition in the following way [4]:

1. Partition the input array into smaller chunks.
2. Have a thread calculate the partial sum for each chunk.
3. Add the partial results from each chunk into a final sum.

That can be implemented with the chunk array being large as the maximum block size and each chunk can be later summarized to the total sum of collisions. The techniques that are used in order to enable this kind of operations and optimize the utilizations of the threads are unrolling loop, in-place reduction and unrolling warps [4]. Those techniques are implemented in the Optimized algorithm with the $block_{cache}$ array being the smaller chunk and the $out_{data}$ array having the values that can later be summarized giving the final sum.

### 4.2.1 Loop unrolling

Loop unrolling technique optimizes the execution of the loop by minimizing the frequency of the branches. It is most effective technique at improving performance for sequential array processing loops where the number of iterations is known prior to execution of the loop[4]. Namely, if the there is an array $a$ with 100 elements and the idea is to sum all the elements from the array then the loop unroll may be implemented. The idea is to, instead of looping 100 times(the length of the array), iterate 50 times, for example, and increase the $sum$ value with $a[i]$ and $a[i+1]$ elements. Thereby, the number of iterations in the enclosing loop is divided by the number of copies made inside the loop body.

### 4.2.2 In-place reduction with unrolling

The maximum thread block size on Maxwell architecture is 1024 and thereby it is possible to completely unroll the loop executions by providing the $block_{cache}$ with the size 1024 which is used in the Optimized algorithm.

### 4.2.3 Unrolling warps

When the number of threads is 32 or less then a single warp is left (warp has 32 or less threads). Considering the SIMT execution of the warps there is an internal synchronization after each instruction and therefore the last 6 iterations of the reduction loop can be

**Algorithm 1** Optimized algorithm using different thread techniques

1: **function** `collisionCounter(` **typrect** $*rectanglesQuery$, **typrect** $*rectanglesData$, **int** $*out_{data}$, **int** $query_{number}$, **int** $data_{number}$)
2: **int** $tid = threadIdx.x$
3: **int** $q_{inx} = blockIdx.x * blockDim.x + threadIdx.x$
4: **int** $sum, d_{inx}$
5: **if** $(idx < query_{number})$ **then**
6: **for** $d_{inx} = 0$ **to** $data_{number}$ **do**
7: **Collision testing**
8: **if** $(data_{rectangle}[d_{inx}][0].h \geq query_{rectangle}[q_{idx}][0].l$ and
$data_{rectangle}[d_{inx}][1].h \geq query_{rectangle}[q_{idx}][1].l$ and
$query_{rectangle}[q_{idx}][0].h \geq data_{rectangle}[d_{inx}][0].l$ and
$query_{rectangle}[q_{idx}][1].h \geq data_{rectangle}[d_{inx}][1].l$ ) **then**
9: $sum+ = 1$
10: **end if**
11: **end for**
12: **end if**
13: **int** $block_{cache}[BLOCKSIZE]$
14: `syncthreads()`
15: **In-place reduction and complete unroll**
16: **if** $(blockDim.x \geq 1024$ and $tid < 512)$ **then**
17: $block_{cache}[tid]+ = block_{cache}[tid + 512]$
18: **end if**
19: `syncthreads()`
20: **if** $(blockDim.x \geq 512$ and $tid < 256)$ **then**
21: $block_{cache}[tid]+ = block_{cache}[tid + 256]$
22: **end if**
23: `syncthreads()`
24: **if** $(blockDim.x \geq 256$ and $tid < 128)$ **then**
25: $block_{cache}[tid]+ = block_{cache}[tid + 128]$
26: **end if**
27: `syncthreads()`
28: **if** $(blockDim.x \geq 128$ and $tid < 64)$ **then**
29: $block_{cache}[tid]+ = block_{cache}[tid + 64]$
30: **end if**
31: `syncthreads()`
32: **Unrolling warps**
33: **if** $(tid < 32)$ **then**
34: **volatile** int $*vsmem = block_{cache}$
35: $vsmem[tid]+ = vsmem[tid + 32]$
36: $vsmem[tid]+ = vsmem[tid + 16]$
37: $vsmem[tid]+ = vsmem[tid + 8]$
38: $vsmem[tid]+ = vsmem[tid + 4]$
39: $vsmem[tid]+ = vsmem[tid + 2]$
40: $vsmem[tid]+ = vsmem[tid + 1]$
41: **end if**
42: **if** $(tid == 0)$ **then**
43: $out_{data}[blockIdx.x] = block_{cache}[0]$
44: **end if**

**Algorithm 2** Non-optimized algorithm algorithm with addAtomic() function

1: **function** `atomCollisionCounter(` **typrect** $*rectanglesQuery$, **typrect** $*rectanglesData$, **int** $query_{number}$, **int** $*counter$, $data_{number})$
2: int $d_{inx}$
3: int $q_{inx} = blockIdx.x * blockDim.x + threadIdx.x$
4: **if** $(idx < query_{number})$ **then**
5: **for** $d_{inx} = 0$ **to** $data_{number}$ **do**
6: **Collision testing**
7: **if** $(data_{rectangle}[d_{inx}][0].h \geq query_{rectangle}[q_{idx}][0].l$ and
$data_{rectangle}[d_{inx}][1].h \geq query_{rectangle}[q_{idx}][1].l$ and
$query_{rectangle}[q_{idx}][0].h \geq data_{rectangle}[d_{inx}][0].l$ and
$query_{rectangle}[q_{idx}][1].h \geq data_{rectangle}[d_{inx}][1].l$ ) **then**
8: `atomicAdd` $(counter, 1)$
9: **end if**
10: **end for**
11: **end if**

unrolled as shown in the pseudo algorithm. The $vsmem$ variable used for the warp unrolling has type **volatile** in order for the compiler not to optimize it and ensure the value is stored.

In the code the `syncthreads()` function is used for the internal block synchronization ensuring that all the treads writing the results to global memory have completed. When it comes to the output that two algorithms produce, the Optimized algorithm returns an array $outdata$ with the values that should later be summed up after the function return in order to have a total number of collisions. On the other hand, Non-optimized algorithm returns a variable $counter$ that has been incremented each time the collision occurred. This distinction in the returning values affected the way of measuring the time as well which is described in the following chapter.

# Chapter 5

# Experimental Evaluation

In this chapter a reader gets familiarized with the actual experiment of the report. There are 7 different test cases of this experiment, measuring the performance of different algorithms and thread organisations. Experimental setup and performance measurements are discussed in this part of the report.

## 5.1 Experimental setup

As in this project the datasets require around 2,2 million threads (1888012 from $data\,rea02$ and 188801 from $query\,rea02$ files), the kernel executions have been launched using one-dimensional blocks in one-dimensional grid. The reason for that lies in the fact that the maximum number of threads per block is 1024 while the first dimension of the grid has its maximum at 2147483647 threads according to `checkDeviceInfor.cu` [4]. The `checkDeviceInfor.cu` program shows the characteristics of the graphics card installed inside a system, in this case NVIDIA GeForce 860m GTX (Appendix A). Considering that this experiment included datasets that together lies around 2,2 million objects which is less than 2147483647 that implies that one-dimensional grids can be implemented by having numerous one-dimensional blocks.

The experiment consists of 7 different test cases. Test cases are run on different percentage of the $rea02\,data$ set starting from 10 and ending with the whole dataset and for each percentage of data the computations are repeated in order to find an average value due to the eventual sources of error. Throughout the experiment the spatial joins are carried out between a particular percentage of the $rea02\,data$ set and the total $rea02\,query$ dataset. The percent defines the amount of randomly chosen rectangles from the $rea02\,data$. To generate the same random sequence of the indexes,$d_{inx}$, the C function `srand()` is implemented to seed the random number generator that is then used by `rand()` function.

Seed number chosen is 10.

The first test case of the experiment consists of implementing the spatial join on the CPU to have a relevant measurement for the number of collisions between the $rea02\,query$ and $rea02\,data$ sets. In order to calculate the mean value for the execution times the test is executed 2 times.

After that, second test case includes the comparison between two different algorithms, both Optimized and Non-optimized algorithm. This test differs from others because it is run on 10 percent of the dataset as its main purpose is to define which is the faster one that is later used for the further test cases. The test starts with 64 threads per block and then doubles until reaching the hardware maximum of 1024, as stated before in the chapter Maxwell architecture constrains the block size to 1024. The usual starting point would be 32 because that is the actual warp size however the Optimized algorithm uses unrolling warps method that needs more than one warp in order to calculate the accurate data. If the block size would be 32 then $vsmem[tid]$ variable would increment with $vsmem[tid + 32]$ and because $tid$ variable indexes the threads inside the block the index would be beyond 32(block size) which causes the data to be incorrect.

The remaining 5 test cases are executed on the whole dataset, staring with 10 and ending with 100 percent. Each of them is executed 5 times to obtain a mean value for the execution time. The difference between them is in the number of threads per block. Starting value for the block size is 64 and is doubled until the 1024 number is reached and thus there are 5 different test cases.

The experiment is conducted on the computer with Ubuntu 16.04 operating system (Appendix A). The compiler used in this project to execute the programs is NVIDIA CUDA compiler, release 8.0, V8.0.61 without any optimization flags included.

## 5.2 Measuring performance

For the benchmarks to be measured the package `<sys/time.h>` have been used by implementing a struct data type **timeval**. On the CPU side, the time that has been measured included only the actual computation of the spatial join because there was no need for either pre-or post-processing. On the other side, the GPU algorithms differed in the way of measuring the time. The Optimized algorithm included the time for sending the data between the CPU and GPU, actual computation of the spatial join, sending back the data to the CPU and the processing of the $out_{data}$ array that included the sum off all $outdata$ array values. Benchmarking the Non-optimized algorithm included sending, computation and receiving time without the extra processing as the $counter$ variable already has the total number of collisions.

# Chapter 6

# Results and Analysis

In this section the results from experimental evaluation are presented and analysed. First the results are visualized using graphs. Then the overall analysis is conducted for the different scenarios, comparing CPU and the faster algorithm on the GPU.

## 6.1 Results

The results are taken from 7 different test cases each of them having several benchmarks. First, the CPU execution times are given showing the execution times for each benchmark. Throughout the experiment, the benchmarks from the same test case have the same thread configurations and are executed on the same percentage of the $rea02data$ set in order to calculate the average execution time. Then the Optimized and Non-optimized algorithms are compared on the GPU in order to determine the most efficient one. After the faster algorithm have been chosen, 5 remaining different test cases are implemented with the faster algorithm including different number of threads per block while executing on 10 to 100 percent of the $rea02data$ set. The measurements are in seconds for the different thread organizations and data percentages. Each GPU benchmark has included time for sending the data from CPU to GPU and back. However, Non-optimized algorithm also includes the time for the post-processing of the returned $out_{data}$ array that is being iterated over and the values are summed up for the final collision counter. More details about the execution times, mean values and coefficient of deviation from the mean values in the Appendix C.

### 6.1.1 Spatial joins on the CPU

The starting point of the experimental evaluation consisted of measuring the spatial join performances on the CPU. Time includes only the actual processing of the spatial joins as no pre- or post-processing is needed. Due to the the high execution times only 2 benchmarks are carried out and the results are visualized in Figure 6.1. Two benchmarks have quite the same performances and thereby they do not significantly differ from each other. The coefficient of variation from the mean value for those two benchmarks is mostly less than 0.1 percent for different percentages of the $rea02data$ set.

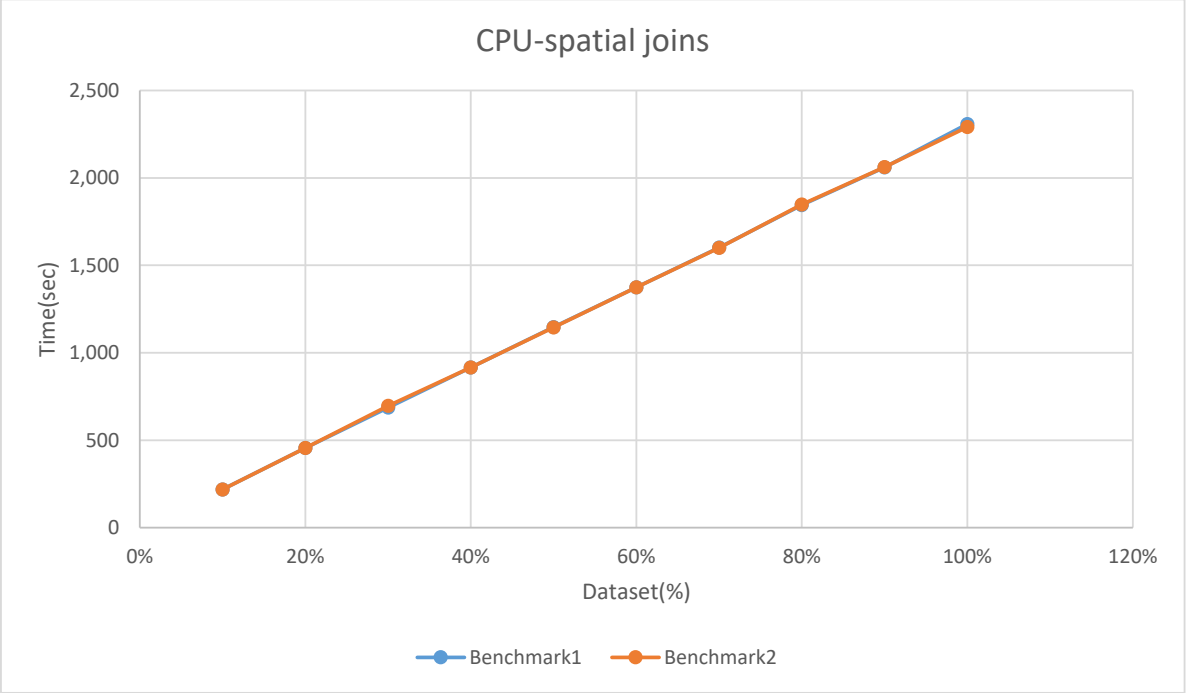


Figure 6.1: CPU execution times

### 6.1.2 GPU spatial joins comparing Optimized and Non-optimized algorithms

After the CPU measurement have been obtained, the performance test for the Optimized and Non-Optimized algorithms is conducted. In this test case both algorithms are tested on 10 percent of the $rea02data$ set in order to fast determine which is the most efficient one for the further tests. Starting blocks size is 64 and it doubles until the number 1024 is reached. According to the Figure 6.2 the Optimized is around 80 percent faster than its counterpart Non-optimized when comparing their mean values. The average execution time for Optimized on the 10 percent of the $rea02data$ set for different block sizes is 4,79 sec and for the Non-optimized it is 8.6 sec.

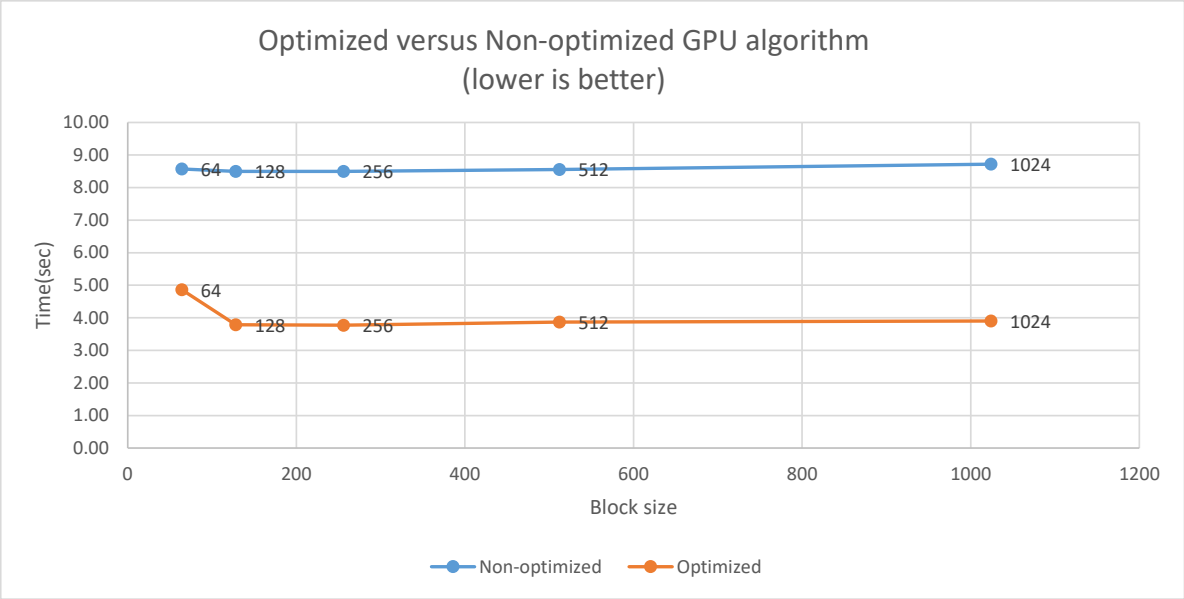


Figure 6.2: Optimized versus Non-optimized

### 6.1.3 GPU spatial joins with block size 64

As the previous measurements showed that the faster algorithm is Optimized it is used for the further progress. In this test case the Optimized algorithm is tested on different percentages of the $rea02data$ set. The block size, i.e. number of threads per block is 64 and there are five benchmarks running on the same configurations for calculating the average values. The execution times for respective benchmark on a given data percentage are shown in the Figure 6.3. The coefficient of deviation from the mean value lies between 0.03 to 4 percent for respective benchmark.

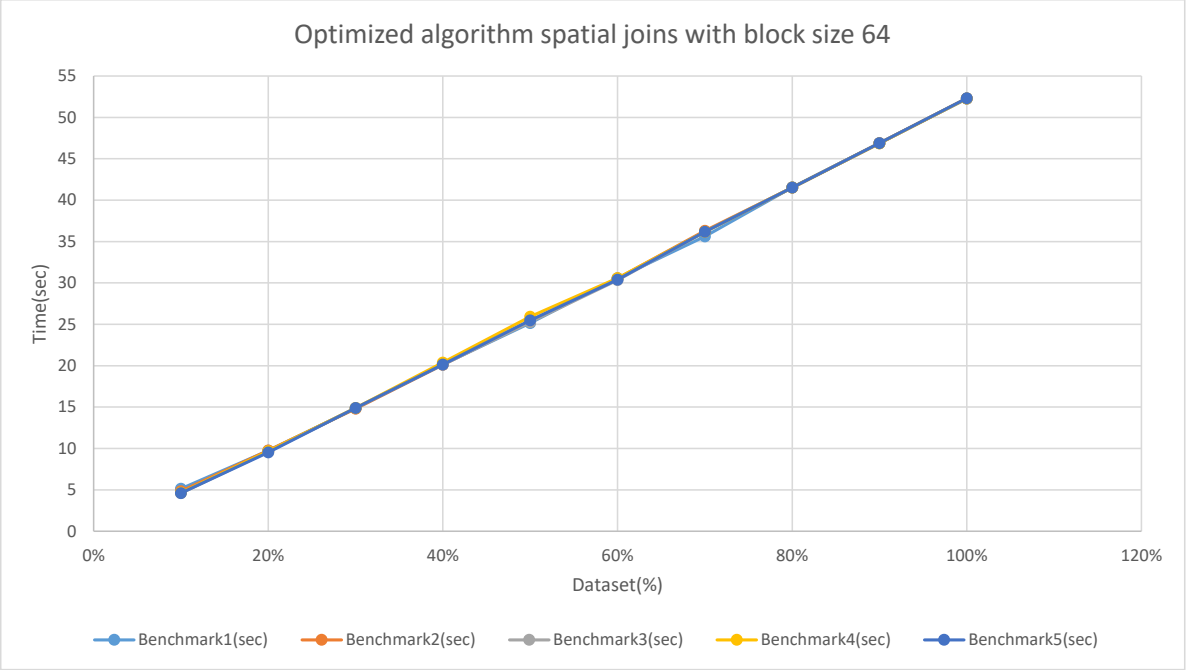


Figure 6.3: Optimized with 64 threads per block

### 6.1.4 GPU spatial joins with block size 128

Next test case includes the Optimized being tested on different percentages of the $rea02data$ set with the block size 128. Again, there are five benchmarks running on the same configurations to obtain an average value. The execution times for respective benchmark on a given data percentage are shown in the Figure 6.4. The coefficient of deviation from the mean value lies between 1 and 5.8 percent for respective benchmark.

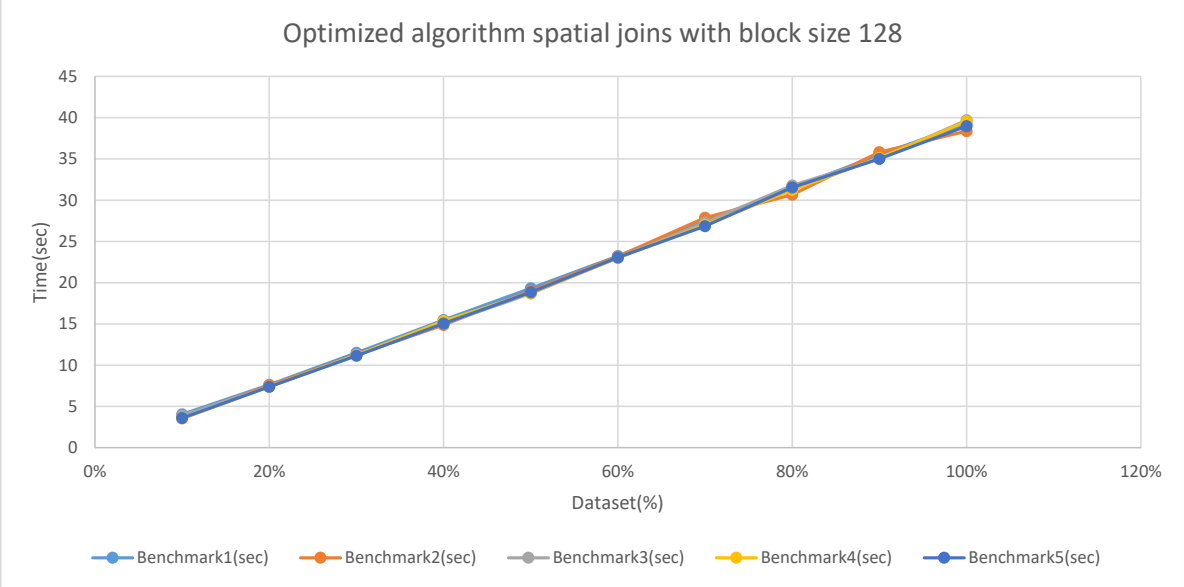


Figure 6.4: Optimized with with 128 threads per block

### 6.1.5 GPU spatial joins with block size 256

Experiment continues with the Optimized algorithm being tested on different percentages of the $rea02data$ set, this time with 256 threads per block. Again, there are five benchmarks running on the same configurations in order to obtain the average execution values. The execution times for respective benchmark on a given data percentage are shown in the Figure 6.5. The coefficient of deviation from the mean value lies between 0.6 and 4.6 percent for respective benchmark.

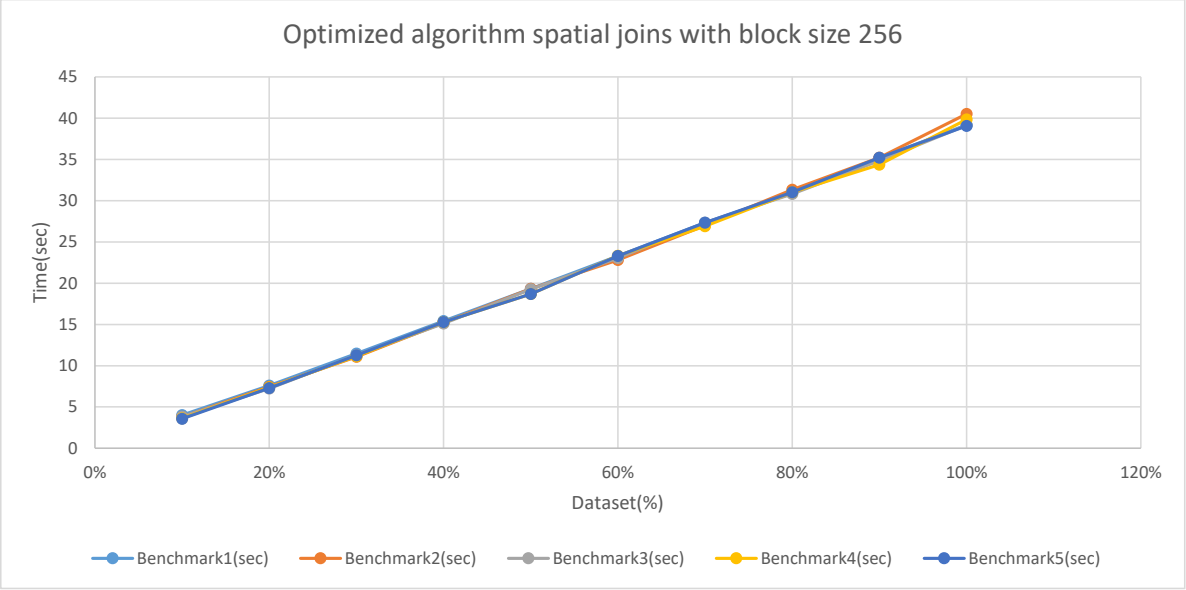


Figure 6.5: Optimized with 256 threads per block

### 6.1.6 GPU spatial joins with block size 512

Sixth test case includes the Optimized algorithm being executed on different percentages of the $rea02data$ set with 512 threads per block. As previously, there are five benchmarks running on the same configurations for calculating the mean values. The execution times for respective benchmark on a specific data percentage are shown in the Figure 6.6. The coefficient of deviation from the mean value is between 0.03 and 4.7 percent for respective benchmark.

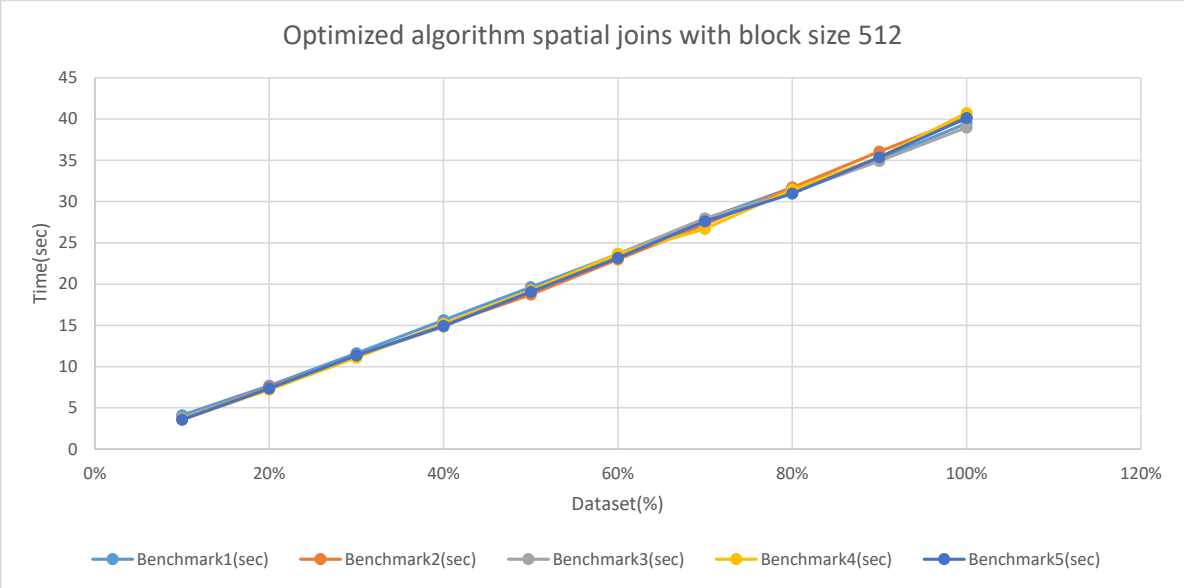


Figure 6.6: Optimized with 512 threads per block

### 6.1.7 GPU spatial joins with block size 1024

The last test case includes the Optimized algorithm being examined on different percentages of the $rea02data$ set with the maximum number of threads per block in a Maxwell architecture. One more time, there are five benchmarks running on the same configurations for obtaining the average values of the execution times. Times for respective benchmark on a specific data percentage are shown in the Figure 6.7. This time, the coefficient of deviation from the mean value lies between 0.6 and 5.8 percent for respective benchmark.

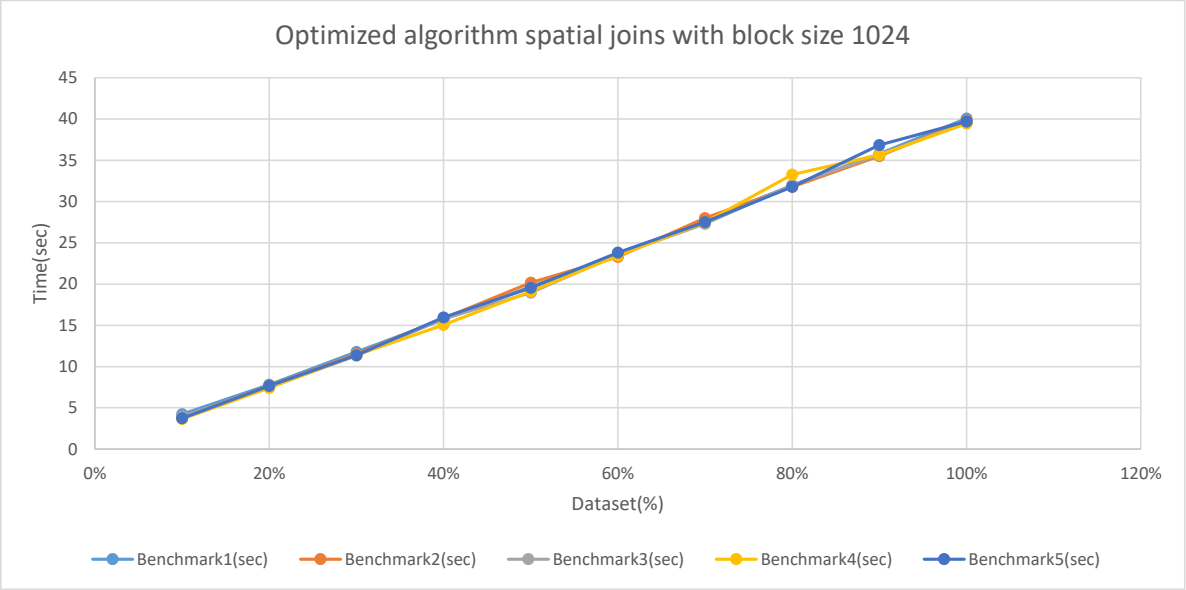


Figure 6.7: Optimized with 1024 threads per block

## 6.2 Validity Analysis

The focus of this project was to decrease the execution time of the spatial joins. As there are two different implementations of the GPU spatial joins that are both faster than the CPU implementation then the resulting work could be classified as valid.

## 6.3 Verification

To verify the GPU implementations, the CPU is used as a relevant measure. Namely, both the GPU and the CPU implementations are handling the spatial join operations. Thereby, if they both have the same functionality, considering that the CPU is the right one, then the GPU can be verified. To verify the implementation on the GPU, each test case shows the output, i.e. number of collided rectangles between $rea02query$ and $rea02data$ set. Those numbers are compared with the number of rectangle collisions from the CPU implementations. According to the values from Appendix C, all of the GPU test cases are showing the same numbers of collided rectangles as those from the CPU implementation. Thereby, as the functionality results the same it means that the GPU implementations are verified.

## 6.4 Analysis and discussion

To get a wider perspective of the previous results the performance analysis and the overall comparison between GPU and CPU implementations are discussed in this section. One feature to notice on the resulting graphs is that the times are increasing linearly. They are proportional to the amount of the data being processed.

### 6.4.1 Optimized vs Non-optimized

In this experiment the two algorithms differ in the way they use threads. Optimized algorithm implements different thread optimization techniques and those are: loop and warp unrolling and in-place reduction. On the other hand, the Non-optimized algorithm implements only the function `addAtomic()` for the threads to synchronize when incrementing the collision counter. Properties of the Optimized algorithm are:

- It is more complex to implement because of the various techniques that are used. Also, it depends on the current block size of the GPU as the lowest number of threads per block is 64.

- The post-processing of the output array $out_{data}$ increases the total time of the computation by summing all of its values in one final collision count variable.
- This Algorithm has shown to be more time efficient as illustrated in the Figure 6.2 and discussed in the Section 5.1.2

When it comes to the Non-optimized algorithm, some of its properties are:

- It is simpler to implement and requires less code. Nevertheless, its function `addAtomic()` is not optimizing the utilization of the threads and this algorithm even works for smaller block size then 64 which is the low limit of algorithm Optimized.
- There is no need for post-processing as the returned value already has the total number of collisions.

Taking into the account the above properties that are discussed the Optimized algorithm is preferable to use because of the better performance even if it is more complex to implement and needs post-processing. Thus, the optimized utilization of the hardware has a significant impact on the performance of the computation, in this case spatial joins.

### 6.4.2 GPU versus CPU

To understand the overall performance increase, the CPU is taken as a relative measure. The Figure 6.8 shows the graph consisting of the execution times for both the CPU and Optimized algorithm for the different percentages of the dataset. The GPU part of the graph consisted of the mean values of all the different block sizes for a specific percentage of the data. The GPU Optimized algorithm speedup is between 54 and 58 times faster than the CPU counterpart. The number of cores inside the CPU and GPU is definitively one of the main reasons for the increase. The current graphics card has 640 cores while the processor used has 4 cores and 8 threads. As the threads are the logical cores of the CPU the expected performance increase could be GPU cores/CPU cores which should result in 80 x speedup. However, the results showed the 54-58x speedup which could be result of the pre-and post-processing of the data, different clock speeds and the GPU warps low occupancy, i.e. if not all the warps are are used throughout the execution [4].

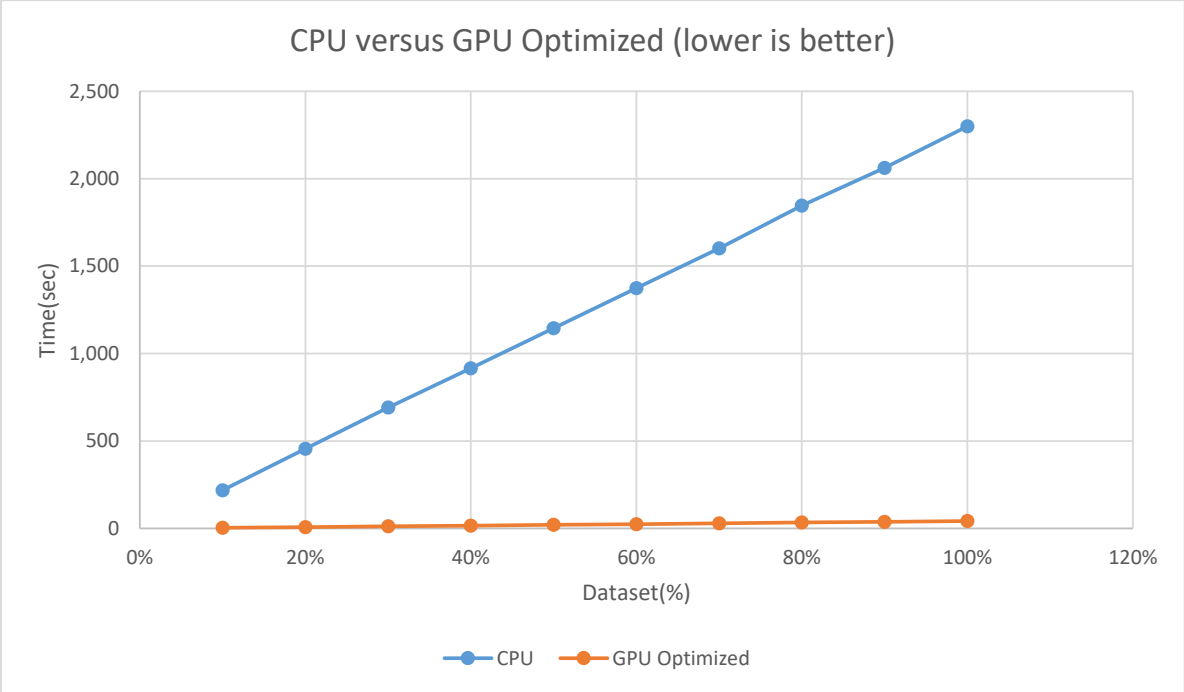


Figure 6.8: Comparing GPU and CPU

### 6.4.3 All the block sizes compared together

By comparing all the test cases for the Optimized algorithm with themselves eventual impact that the block size has on the performance can be revealed. There are five test cases that differ in the number of threads per block. Figure 6.9 shows the average execution time for respective percentage of $rea02data$ set for all of the five different tests. According to the figure the test cases with 128, 256, 512 and 1024 have similar execution times while test case with 64 threads per block has increased execution time. The execution of the Optimized algorithm with the block size at 64 is 30% slower which indicates that even if the block size is as twice large as a single warp (which should be enough for the Optimized algorithm as discussed in the Section 4.3) there is no guarantee that it will execute fast enough as other block sizes.

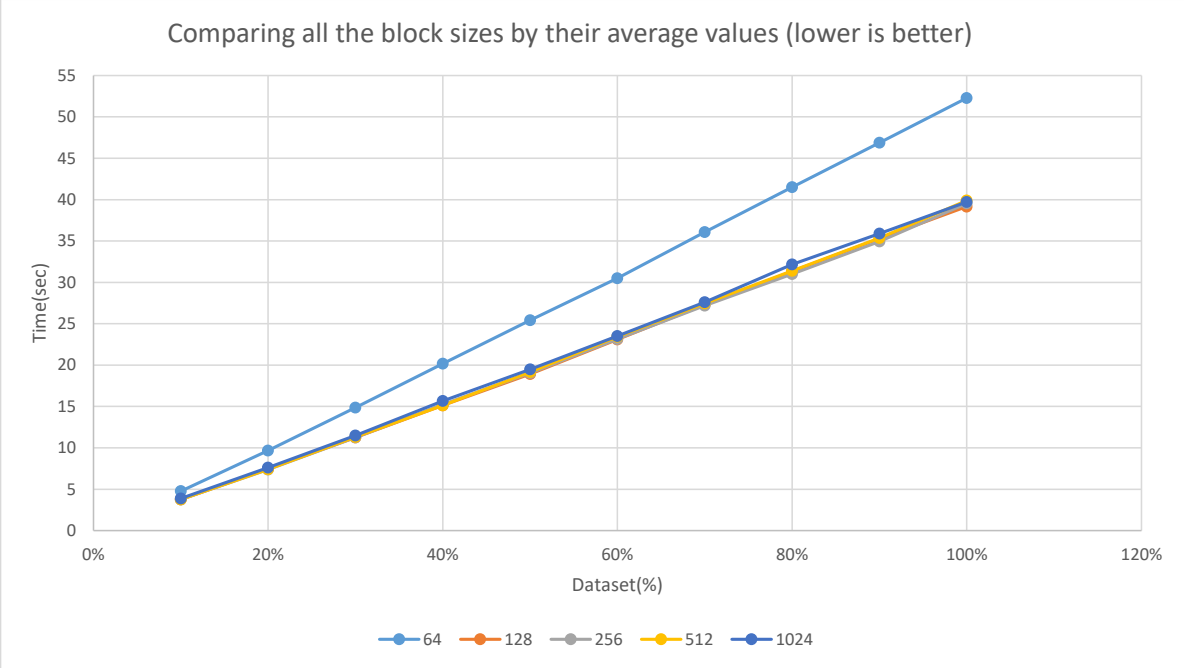


Figure 6.9: Comparing the execution of Optimized algorithm with different block sizes

# Chapter 7

# Conclusion and future work

In this chapter the conclusions about the work are made and future work is discussed with the delimitations pointing out eventual improvement for the future work.

## 7.1 Conclusions

This report covered the spatial joins and their implementation on both GPU and CPU. The problem that this report solved was the potential improvement of thread organizations on the GPU spatial joins operations. Report begins with the background that provided the required knowledge for the models to be created and later used in the experiment. There are also potential error sources that could cause the results to differ from the right ones, however that was the reason why the same test cases were repeated in order to minimize the eventual deviations from the correct values. After the results have been gathered and visualized, following insights were acquired:

- Overall performance impact of the GPU accelerated spatial joins when compared to the CPU counterpart
- The importance of the optimized thread utilization
- The dependency of the thread organizations on different hardware aspect, i.e warp and block size

As expected, number of cores makes a difference when there are data independent calculations, as spatial joins, that can be computed in SIMT fashion. That is the reason why GPU with several hundred cores is 54-58 times faster than the CPU with only several cores. This gives the insight that it is important to know when the program be parallelized in order to increase its performance and be able to utilize the potential that the GPU: s offer.

By optimizing the threads in the computations of spatial joins the performances increased in 80 percent when compared to the Non-optimized algorithm. That gives the programmer an important aspect to consider when developing the software as 80 percent is not imperceptible. Thereby, writing the programs that optimizes the underlying hardware by implementing different thread techniques is a vital aspect to consider when performance is important.

Also, this report showed the dependency that different thread organisations have on the actual GPU installed in a system. Things that are important to consider are: warp size and maximum number of threads per block. Those components are vital when trying to optimize the execution because they present limits of the system being run on. Warps are important to consider as they make it possible to completely unroll the execution of the program. On the other side, block size also affects the actual loop unrolling because its maximum size decides the upper limit for the thread index check.

This project reveals the actual power that the parallel programming and thread optimization have in the intensive computations. This is the ”new dimension” of thinking about programming which opens new views and perspective to the computer science. The very interesting part with this bachelor thesis was that this field is being developed because there are many use cases of the GPU spatial joins such as gaming industry, neuroscience, geographical systems etc. Researching about the GPU for the data-intensive processing reveals the potential that the GPU have in many other areas apart from the visualizations.

## 7.2 Limitations

The potential limitation that are presented in this project is the hardware that the system had. If there were new generations of the graphics card and processor the performances would be significantly better and the actual time for the computations would be much less.

## 7.3 Future work

The future work could include different types of spatial joins algorithms. Namely, spatial subdivision and sweet and sort should be tested as well. Also, different dimensions of the grids and blocks, two-and three-dimensions could be test to see how they affect performance of the spatial joins.

## 7.4 Acknowledgements

I would like to thank Professor Christian Schulte (KTH) on his interesting and motivational ideas that directed the flow of the report. Professor Schulte was both the examiner and supervisor for the thesis and was always interested to help and discuss the ongoing process and solutions for the eventual barriers. When it comes to the related materials about the subject i would like to mention that Professor Gerald Maguire (KTH) provided related articles and papers.

# .1 Appendix A - System Specifications

The system used in this project consisted of Intel® Core™ i7-4710HQ processor, 16 GB DDR3 RAM running on Ubuntu 16.04 Operating System with NVIDIA GTX 860m (Maxwell version) [17] graphics card [1]:

Table 1: NVIDIA GeForce 860m GTX specifications

| | |
|---|---|
| GPU | GM107 |
| SMM | 5 |
| ALUs | 128 |
| CUDA Cores | 640 |
| Technology | 28 nm |
| Release Date | Mar 12, 2014 |
| Transistors | 1870M |
| Graphics Clock | 540 MHZ |
| Memory Clock | 1253 MHz |
| Memory Size | 2048 MB |
| Memory Type | GDDR5 |
| Bus Width | 128 Bit |
| Memory Bandwidth | 80.2 GB/s |
| Computing | OpenCL, CUDA, PhysX, Direct Compute 5.0 |
| Raster Operations | 16 |
| Bus Interface | PCI-E 3.0 x 16 @x8.1.1 |
| DirectX Support | 11.0 |
| Memory Type | GDDR5 |
| Pixel Filtrate | 8.6 GPixel/s |
| Texture Filtrate | 21.6 G Texel/s |

# .2 Appendix B

GTX 860m operates on Five SMM where each SMM has 128 ALUs. The GigaThread engine (shown in orange on the left side of the diagram) is a global scheduler that distributes thread blocks to the SM warp schedulers [4]. Warp schedulers are inside the SMs and they schedule the warps that are processed by different SMs because the SMs have limited resources.

[1] TechPowerUp GPU-Z 1.20.0

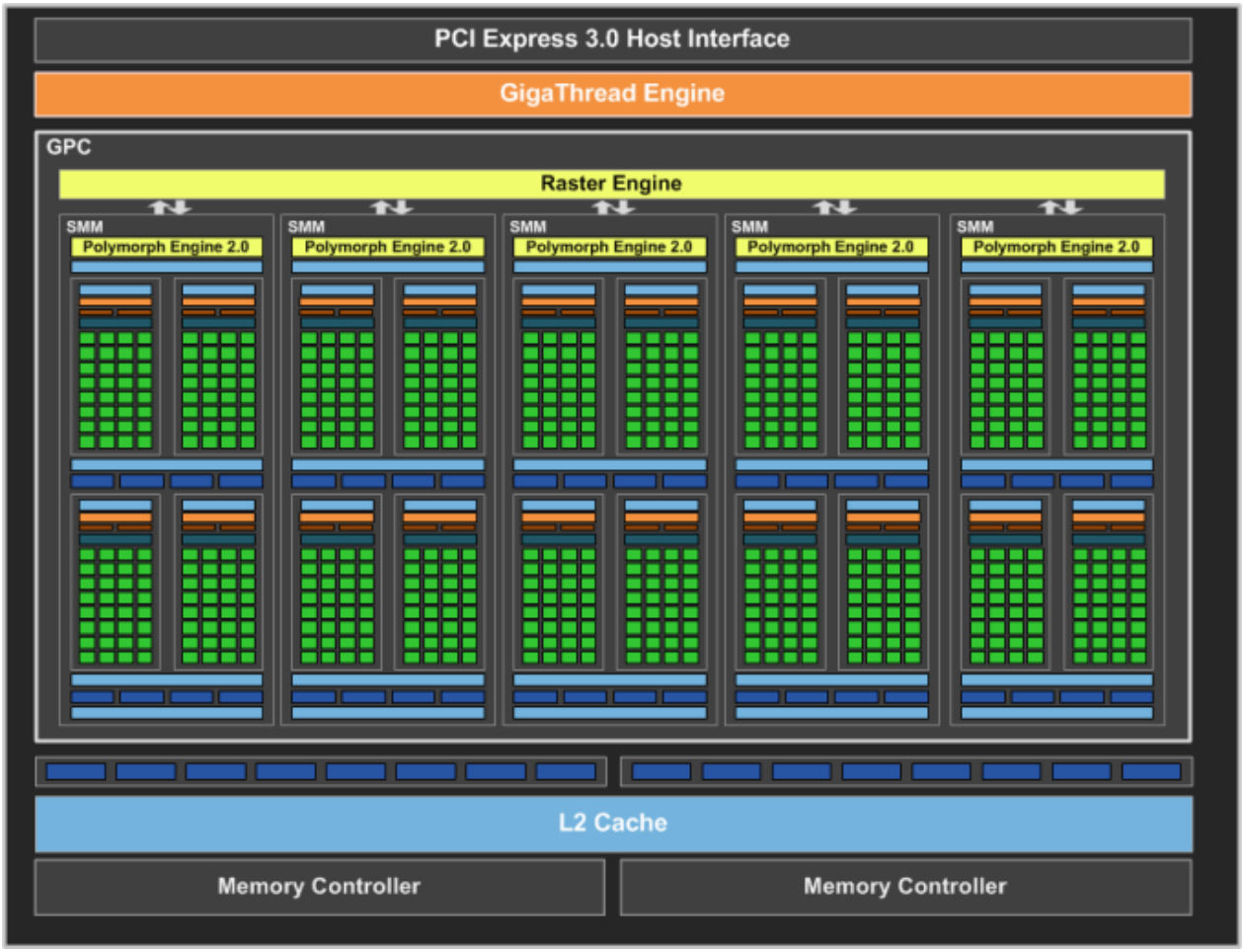


Figure 1: Five SMM architecture of NVIDIA GTX 860m Graphics Card [18]

# .3 Appendix C

All the details about the execution times are presented below. Mean shows the average execution times while the CV presents the coefficient of deviation and counter provides the number of intersected objects between $rea02query$ and $rea02data$ sets.

Table 2: Optimized vs Non-optimized algorithm on 10% of the $rea02\,data$ set

| Block sizes | 64 | 128 | 256 | 512 | 1024 | Mean | CV | Counter |
|---|---|---|---|---|---|---|---|---|
| Non-optimized algorithm | 8.57 | 8.50 | 8.50 | 8.56 | 8.72 | 8.59717 | 1.234733 | 22682 |
| Algotihm 1 | 4.87 | 3.78 | 3.77 | 3.87 | 3.91 | 4.791067 | 39.419781 | 22682 |

Table 3: CPU execution times

| Data percentage | B1(sec) | B2(sec) | Mean | CV(%) | Counter |
|---|---|---|---|---|---|
| AA 10% | 218.442 | 218.314518 | 218.378259 | 0.041229 | 22682 |
| 20% | 455.935 | 456.085385 | 456.0101925 | 0.023275 | 45138 |
| 30% | 685.806 | 697.466274 | 691.636137 | 1.192065 | 67903 |
| 40% | 915.013 | 915.139006 | 915.076003 | 0.009745 | 91553 |
| 50% | 1145.947 | 1145.248217 | 1145.597609 | 0.043155 | 113610 |
| 60% | 1373.57 | 1374.196426 | 1373.883213 | 0.032263 | 136350 |
| 70% | 1601.977 | 1600.357382 | 1601.167191 | 0.071516 | 159255 |
| 80% | 1844.471 | 1847.037494 | 1845.754247 | 0.098307 | 181957 |
| 90% | 2061.033 | 2062.200432 | 2061.616716 | 0.040036 | 203900 |
| 100% | 2308.029 | 2290.827586 | 2299.428293 | 0.528974 | 227798 |

Table 4: Optimized algorithm with 64 threads per block

| Data | B1(sec) | B2(sec) | B3(sec) | B4(sec) | B5(sec) | Mean | CV(%) | Collisions |
|---|---|---|---|---|---|---|---|---|
| 10% | 5.13 | 4.83 | 4.61 | 4.65 | 4.60 | 4.765462 | 4.739846 | 22682 |
| 20% | 9.73 | 9.76 | 9.62 | 9.68 | 9.53 | 9.662328 | 0.942137 | 45138 |
| 30% | 14.86 | 14.80 | 14.89 | 14.88 | 14.88 | 14.859958 | 0.255706 | 67903 |
| 40% | 20.14 | 20.11 | 20.11 | 20.37 | 20.13 | 20.172268 | 0.552028 | 91553 |
| 50% | 25.15 | 25.39 | 25.28 | 25.92 | 25.46 | 25.440121 | 1.149336 | 113610 |
| 60% | 30.57 | 30.55 | 30.35 | 30.59 | 30.41 | 30.492176 | 0.355233 | 136350 |
| 70% | 35.61 | 36.32 | 36.14 | 36.18 | 36.17 | 36.084473 | 0.752963 | 159255 |
| 80% | 41.54 | 41.51 | 41.51 | 41.53 | 41.52 | 41.524242 | 0.032307 | 181957 |
| 90% | 46.89 | 46.90 | 46.87 | 46.87 | 46.90 | 46.884201 | 0.029741 | 203900 |
| 100% | 52.25 | 52.30 | 52.29 | 52.28 | 52.31 | 52.285881 | 0.043919 | 227798 |

Table 5: Optimized algorithm with 128 threads per block

| Data | B1(sec) | B2(sec) | B3(sec) | B4(sec) | B5(sec) | Mean | CV(%) | Collisions |
|---|---|---|---|---|---|---|---|---|
| 10% | 4.05 | 3.75 | 3.90 | 3.52 | 3.58 | 3.761014 | 5.850166 | 22682 |
| 20% | 7.61 | 7.56 | 7.42 | 7.39 | 7.37 | 7.469688 | 1.472209 | 45138 |
| 30% | 11.51 | 11.25 | 11.23 | 11.22 | 11.13 | 11.268587 | 1.26616 | 67903 |
| 40% | 15.46 | 14.87 | 15.00 | 15.36 | 15.05 | 15.147425 | 1.64811 | 91553 |
| 50% | 19.31 | 19.03 | 18.68 | 18.74 | 18.83 | 18.920513 | 1.356639 | 113610 |
| 60% | 23.24 | 23.17 | 23.01 | 22.98 | 23.04 | 23.087584 | 0.488465 | 136350 |
| 70% | 27.46 | 27.88 | 27.29 | 27.01 | 26.85 | 27.297039 | 1.481921 | 159255 |
| 80% | 31.46 | 30.65 | 31.77 | 31.24 | 31.55 | 31.333933 | 1.36613 | 181957 |
| 90% | 35.51 | 35.82 | 35.05 | 35.12 | 35.01 | 35.301048 | 0.990546 | 203900 |
| 100% | 39.27 | 38.37 | 39.69 | 39.55 | 38.97 | 39.167778 | 1.343267 | 227798 |

Table 6: Optimized algorithm with 256 threads per block

| Data | B1(sec) | B2(sec) | B3(sec) | B4(sec) | B5(sec) | Mean | CV(%) | Collisions |
|---|---|---|---|---|---|---|---|---|
| 10% | 4.01 | 3.79 | 3.80 | 3.63 | 3.57 | 3.760193 | 4.615478 | 22682 |
| 20% | 7.58 | 7.43 | 7.22 | 7.37 | 7.29 | 7.378388 | 1.867346 | 45138 |
| 30% | 11.47 | 11.08 | 11.30 | 11.15 | 11.25 | 11.250841 | 1.33568 | 67903 |
| 40% | 15.42 | 15.21 | 15.11 | 15.28 | 15.28 | 15.258107 | 0.738173 | 91553 |
| 50% | 19.31 | 19.34 | 19.25 | 18.67 | 18.70 | 19.053783 | 1.766077 | 113610 |
| 60% | 23.29 | 22.81 | 23.06 | 23.32 | 23.29 | 23.155262 | 0.942095 | 136350 |
| 70% | 27.31 | 27.01 | 27.33 | 26.91 | 27.34 | 27.178696 | 0.751792 | 159255 |
| 80% | 30.90 | 31.33 | 30.81 | 31.04 | 31.04 | 31.020741 | 0.640383 | 181957 |
| 90% | 35.14 | 35.20 | 34.86 | 34.36 | 35.22 | 34.955345 | 1.043482 | 203900 |
| 100% | 39.18 | 40.51 | 39.14 | 39.84 | 39.06 | 39.545197 | 1.57746 | 227798 |

Table 7: Optimized algorithm with 512 threads per block

| Data | B1(sec) | B2(sec) | B3(sec) | B4(sec) | B5(sec) | Mean | CV(%) | Collisions |
|---|---|---|---|---|---|---|---|---|
| 10% | 4.12 | 3.78 | 3.81 | 3.58 | 3.55 | 3.767646 | 5.980144 | 22682 |
| 20% | 7.69 | 7.51 | 7.34 | 7.18 | 7.36 | 7.415374 | 2.580137 | 45138 |
| 30% | 11.63 | 11.26 | 11.30 | 11.09 | 11.35 | 11.324651 | 1.734150 | 67903 |
| 40% | 15.63 | 15.06 | 14.83 | 15.19 | 14.93 | 15.127666 | 2.048525 | 91553 |
| 50% | 19.62 | 18.71 | 19.23 | 19.24 | 19.06 | 19.171654 | 1.717196 | 113610 |
| 60% | 23.63 | 22.99 | 23.64 | 23.64 | 23.18 | 23.414497 | 1.319109 | 136350 |
| 70% | 27.88 | 27.24 | 27.97 | 26.68 | 27.60 | 27.471279 | 1.914936 | 159255 |
| 80% | 31.69 | 31.71 | 31.12 | 31.46 | 30.97 | 31.38995 | 1.059426 | 181957 |
| 90% | 35.09 | 36.05 | 34.92 | 35.37 | 35.36 | 35.355682 | 1.222242 | 203900 |
| 100% | 39.47 | 40.20 | 38.98 | 40.73 | 40.11 | 39.900791 | 1.704408 | 227798 |

Table 8: Optimized algorithm with 1024 threads per block

| Data | B1(sec) | B2(sec) | B3(sec) | B4(sec) | B5(sec) | Mean | CV(%) | Collisions |
|---|---|---|---|---|---|---|---|---|
| 10% | 4.23 | 3.88 | 3.94 | 3.65 | 3.73 | 3.884006 | 5.796392 | 22682 |
| 20% | 7.80 | 7.66 | 7.50 | 7.42 | 7.64 | 7.603368 | 1.904543 | 45138 |
| 30% | 11.77 | 11.50 | 11.33 | 11.40 | 11.39 | 11.478873 | 1.505135 | 67903 |
| 40% | 15.73 | 15.83 | 15.77 | 15.04 | 15.95 | 15.663025 | 2.300148 | 91553 |
| 50% | 19.68 | 20.17 | 18.96 | 19.10 | 19.54 | 19.487606 | 2.486947 | 113610 |
| 60% | 23.46 | 23.30 | 23.57 | 23.44 | 23.80 | 23.513649 | 0.784832 | 136350 |
| 70% | 27.71 | 27.99 | 27.29 | 27.48 | 27.52 | 27.598415 | 0.96196 | 159255 |
| 80% | 31.96 | 31.78 | 32.00 | 33.28 | 31.78 | 32.159641 | 1.979565 | 181957 |
| 90% | 35.77 | 35.53 | 35.66 | 35.68 | 36.84 | 35.895153 | 1.489889 | 203900 |
| 100% | 40.07 | 39.83 | 39.56 | 39.45 | 39.70 | 39.720295 | 0.610264 | 227798 |

Table 9: Comparing all the test cases with different block sizes of the Optimized algorithm by their average execution times

| Data | 64 | 128 | 256 | 512 | 1024 | Mean |
|---|---|---|---|---|---|---|
| 10% | 4.77 | 3.76 | 3.76 | 3.77 | 3.88 | 3.99 |
| 20% | 9.66 | 7.47 | 7.38 | 7.42 | 7.60 | 7.91 |
| 30% | 14.86 | 11.27 | 11.25 | 11.32 | 11.48 | 12.04 |
| 40% | 20.17 | 15.15 | 15.26 | 15.13 | 15.66 | 16.27 |
| 50% | 25.44 | 18.92 | 19.05 | 19.17 | 19.49 | 20.41 |
| 60% | 30.49 | 23.09 | 23.16 | 23.41 | 23.51 | 24.73 |
| 70% | 36.08 | 27.30 | 27.18 | 27.47 | 27.60 | 29.13 |
| 80% | 41.52 | 31.33 | 31.02 | 31.39 | 32.16 | 33.49 |
| 90% | 46.88 | 35.30 | 34.96 | 35.36 | 35.90 | 37.68 |
| 100% | 52.29 | 39.17 | 39.55 | 39.90 | 39.72 | 42.12 |

Table 10: Comparing the mean values for both Optimized algorithm and CPU

| Data | CPU | Optimized algorithm |
|---|---|---|
| 10% | 218.378259 | 3.987667 |
| 20% | 456.0101925 | 7.90579744 |
| 30% | 691.636137 | 12.0365792 |
| 40% | 915.076003 | 16.2737408 |
| 50% | 1145.597609 | 20.4147396 |
| 60% | 1373.883213 | 24.7326312 |
| 70% | 1601.167191 | 29.1259644 |
| 80% | 1845.754247 | 33.48573392 |
| 90% | 2061.616716 | 37.67827224 |
| 100% | 2299.428293 | 42.12401528 |

# Bibliography


[1] O. Gunther, “Efficient computation of spatial joins,” in *Data Engineering, 1993. Proceedings. Ninth International Conference on*, IEEE, 1993, pp. 50–59.

[2] S. Nobari, F. Tauheed, T. Heinis, P. Karras, S. Bressan, and A. Ailamaki, “TOUCH: In-memory spatial join by hierarchical data-oriented partitioning,” in *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, ACM, 2013, pp. 701–712. [Online]. Available: http://dl.acm.org/citation.cfm?id=2463700 (visited on 04/07/2017).

[3] NVIDIA, *About cuda*, https://developer.nvidia.com/about-cuda, Accessed: 2017-04-17, Mar. 6, 2014.

[4] J. Cheng, *Professional cuda c programming*. Indianapolis, IN: John Wiley and Sons, 2014, ISBN: 978-1-118-73932-7.

[5] V. Narasiman, M. Shebanow, C. J. Lee, R. Miftakhutdinov, O. Mutlu, and Y. N. Patt, “Improving gpu performance via large warps and two-level warp scheduling,” in *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-44, Porto Alegre, Brazil: ACM, 2011, pp. 308–317, ISBN: 978-1-4503-1053-6. DOI: 10.1145/2155620.2155656. [Online]. Available: http://doi.acm.org.focus.lib.kth.se/10.1145/2155620.2155656.

[6] IEzEE, *Code of ethics*, https://www.ieee.org/about/corporate/governance/p7-8.html, Accessed: 2017-04-18.

[7] S. Le Grand, *GPU Gems 3 - Chapter 32. Broad-Phase Collision Detection with CUDA*. [Online]. Available: http://http.developer.nvidia.com/GPUGems3/gpugems3_ch32.html (visited on 04/18/2017).

[8] A. Belussi, E. Bertino, and A. Nucita, "Grid based methods for estimating spatial join selectivity," in *Proceedings of the 12th annual ACM international workshop on Geographic information systems*, ACM, 2004, pp. 92–100. [Online]. Available: http://dl.acm.org/citation.cfm?id=1032238 (visited on 04/20/2017).

[9] E. H. Jacox and H. Samet, "Iterative spatial join," *ACM Trans. Database Syst.*, vol. 28, no. 3, pp. 230–256, Sep. 2003, ISSN: 0362-5915. DOI: 10.1145/937598.937600. [Online]. Available: http://doi.acm.org.focus.lib.kth.se/10.1145/937598.937600.

[10] D. Luebke, "Cuda: Scalable parallel programming for high-performance scientific computing," in *2008 5th IEEE International Symposium on Biomedical Imaging: From Nano to Macro*, May 2008, pp. 836–838. DOI: 10.1109/ISBI.2008.4541126.

[11] NVIDIA, "REVOLUTIONIZING HIGH PERFORMANCE COMPUTING NVIDIA TESLA," [Online]. Available: http://www.ts.avnet.com/uk/vendors/nvidia/assets/brochure_june12.pdf (visited on 04/30/2017).

[12] *CUDA 8 Features Revealed: Pascal, Unified Memory and More*, 2016. [Online]. Available: https://devblogs.nvidia.com/parallelforall/cuda-8-features-revealed/ (visited on 04/26/2017).

[13] S. Ryoo, C. I. Rodrigues, S. S. Baghsorkhi, S. S. Stone, D. B. Kirk, and W.-m. W. Hwu, "Optimization principles and application performance evaluation of a multithreaded GPU using CUDA," in *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, ACM, 2008, pp. 73–82. [Online]. Available: http://dl.acm.org/citation.cfm?id=1345220 (visited on 04/17/2017).

[14] A. Håkansson, “Portal of research methods and methodologies for research projects and degree projects,” in *Proceedings of the International Conference on Frontiers in Education: Computer Science and Computer Engineering (FECS)*, The Steering Committee of The World Congress in Computer Science, Computer Engineering and Applied Computing (WorldComp), 2013, p. 1.

[15] R. L. Goodstein, *Boolean algebra*. Courier Corporation, 2007.

[16] N. Beckmann and B. Seeger, *A benchmark for multidimensional index structures*. [Online]. Available: `http://www.mathematik.uni-marburg.de/~rstar/benchmark/distributions.pdf`.

[17] NVIDIA, *GeForce GTX 860m — Specifications — GeForce*. [Online]. Available: `http://www.geforce.com/hardware/notebook-gpus/geforce-gtx-860m/specifications` (visited on 05/03/2017).

[18] *NVIDIA GeForce GTX 860m Features Maxwell GM107 Core and 45w TDP - Benchmarks Unveiled*, Mar. 2014. [Online]. Available: `http://wccftech.com/nvidia-geforce-gtx-860m-features-maxwell-gm107-core-45w-tdp-benchmarks-unveiled/` (visited on 05/04/2017).

TRITA TRITA-ICT-EX-2017:74