TALLINN UNIVERSITY OF TECHNOLOGY
Faculty of Information Technology
Department of Software Science

Andrei Orehhov 132668 IAPM

USING CUDA FOR SPEEDING UP IN-MEMORY DATABASES WITH THE HELP OF GRAPHICS COPROCESSORS

Master's thesis

Supervisor: Tanel Tammet
Professor

Tallinn 2017

TALLINNA TEHNIKAÜLIKOOL
Infotehnoloogia teaduskond
Tarkvarateaduse instituut

Andrei Orehhov 132668 IAPM

CUDA RAAMISTIKU KASUTAMINE MÄLUANDMEBAASIDE TÖÖ KIIRENDAMISEKS GRAAFIKAPROTSESSORI ABIL

Magistritöö

Juhendaja: Tanel Tammet
Professor

Tallinn 2017

Author’s declaration of originality

I hereby certify that I am the sole author of this thesis. All the used materials, references to the literature and the work of others have been referred to. This thesis has not been presented for examination anywhere else.

Author: Andrei Orehhov

[dd.mm.yyyy]

Abstract

In this thesis we will investigate the ways to use a graphic processing unit (GPU) to improve data querying capabilities in the context of in-memory databases. At the beginning of the thesis, we outline the basics of GPU programing using a parallel computing platform known as CUDA. We explore examples of successful integration of CUDA with different Database Management Systems (DBMSs). With acquired knowledge, we define and code in C/C++ programing language a set of experimental scenarios that will enable us to understand CUDA behaviour in regards to memory allocation and threads manipulation. Using obtained results, we will highlight the advantages and disadvantages of using GPU memory vs RAM memory for the task of querying data. Finally, we attempt to integrate CUDA with an existing lightweight NoSql database library (WhiteDB) which was developed in Tallinn University of Technology and present the results and conclusions.

This thesis is written in English and is 55 pages long, including 11 chapters, 11 figures and 9 tables.

Annotatsioon

CUDA raamistiku kasutamine mäluandmebaaside töö kiirendamiseks graafikaprotsessori abil

Käesolevas magistritöös uuritakse võimalusi graafikakaardi (GPU) kasutamiseks mälupõhiste andmebaaside päringute kiirendamiseks. Töö alguses tutvustame GPU programmeerimise põhitõdesid CUDA raamistikus ning kirjeldame GPU rakendamise näiteid erinevates andmebaasisüsteemides. Seejärel defineerime mitu eksperimentaalstsenaariumit CUDA kasutusvõimaluste ja efektiivsuse uurimiseks ning realiseerime nad C ++ programmeerimiskeeles. Eksperimenteerimise fookuses on mälukasutus ja lõimed. Tulemuste baasil anname kokkuvõtte probleemidest ja võimalikest kiirusevõitudest. Töö lõpuks katsetame CUDA integreerimist olemasoleva NoSQL andmebaasiteegiga WhiteDB ning toome välja keerukusi ja takistusi, mis sellise integreerimise jaoks tuleks lahendada.

Lõputöö on kirjutatud inglise keeles ning sisaldab 55 lehekülge teksti, 11 peatükki, 11 joonist, 9 tabelit.

List of abbreviations and terms

CPU	*Central processing unit*
GPU	*Graphics processing Unit*
PS	Personal computer
RAM	*Random-access memory*
SM	*Streaming multiprocessor*
MMDBMS	*Main Memory Database Management System*
DRDBMS	*Distributed Relational Database Management Systems*
RDBMS	*Traditional Database Management System*
SQL	*Structured Query Language*
PCI	*Peripheral component interconnect*
FDTD	*Finite-Difference Time-Domain*
OLTP	*Online Transaction Processing*

Table of contents

List of figures

List of tables

1 Introduction

Graphics processing units (GPUs) are used today in a wide range of applications, mainly because they can dramatically accelerate parallel computing, are affordable and energy efficient. Driven by the market demand for real-time, high-definition 3D graphics, the programmable GPU has evolved into highly parallel, multithreaded, many core processors [1]. The desire to use GPU as a more general parallel computing device motivated NVIDIA to develop a new unified graphics and computing GPU architecture and the CUDA programming model [1]. CUDA programming model and software environment let programmers write scalable parallel programs using a straightforward extensions of the C language [2]. In the years since its release, many developers have used CUDA to parallelize and accelerate computations across various problem domains [2].

Since high performance is a crucial part of database processing, there is constant research how CUDA can increase data mining capabilities. It has been proven that executing queries through the GPU virtual machine using CUDA can provide faster results in contrast to, for example, CPU SQLite virtual machine [3].

1.1 Goal

The goal of the thesis to understand the basics of CUDA programming and highlight all the advantages and disadvantages of CUDA integration for querying in in-memory databases. We will write working CUDA code to experimentally perform search in a created fix-sized dataset which will be represented as a two-dimensional array consisting of 50 million values. The dataset will be filled with integer values. All the experimental programs will simply identify and count all the records that contain a specific targeted value.

All CUDA operations including memory allocation, memory transfer and functions execution must be measured in real time. Results of GPU performance need to be

documented and compared to a standard CPU execution with the same fix-sized dataset. Provided results will identify the level of performance increase enabled by GPU vs CPU execution and determine the best suited scenarios for CUDA implementation.

Finally, based on the acquired knowledge we will attempt to integrate CUDA with WhiteDB. Since full integration with CUDA is a very challenging task we will focus on a few concrete experiments. We will attempt to reproduce the same data arrays which were used in previous experimentations and count all the identified records.

Integration steps:

- Attempt to allocate memory for a local database on GPU using CUDA
- Define and copy data for CUDA local database
- Adapt necessary WhiteDB methods related to SELECT operations in CUDA
- In case of successful implementation measure GPU execution time and compare to CPU

1.2 Background

In the modern world we always encounter challenges in regards to performance of our systems. The increasing volume of data (big data) generated by entities unquestionably, require high performance parallel processing models for robust and speedy data analysis [4]. Modern processing architectures exploit parallelism on a number of levels, including instruction-level parallelism, multitasking, and multithreading [5].

For demonstration purposes, let's assume we have three arrays of integer type – array A, array B and C consisting of 100 random integer values. By defined scenario, values from A and B must be multiplied by their respective index and resulting array will be stored in array C.

Solution in this case is very trivial. Programmers are required to write a standard "for" or "while" loop that will go through all the values and multiply by their respective index. End result is stored in a resulting array by their respective index as well. However, this

approach highlights the main drawback. Just like in any CPU program the process is sequential – we go through value after value which logically consumes time.

Index	Array A	Array B	Multiplication	Array C
0	1	6	→	6
1	7	2	→	14
2	9	1	→	9
3	10	3	→	30
4	15	2	→	30
5	4	5	→	20
…	…	…	→	…

Table 1:Sequential multiplication

It would have been beneficial if we have executed a prepared number of independent threads for each same indexed values from both arrays and have them executed at the same time. This approach can significantly boost the scaling performance of the application which can be enabled by integrating our source code with CUDA.

1.3 Summary

The thesis is divided into 4 parts. In first part we are introduced to the CUDA programing model. In the second part we explain how CUDA can have a relationship with database management systems. Third part introduces us to a set of experimentation scenarios with CUDA and CUDA integration attempts with NoSql library WhiteDB. Fourth part deals with analysis of all the experiments. Thesis ends with a conclusion and summary.

2 CUDA overview

Since NVIDIA released CUDA in 2007, developers have rapidly developed scalable parallel programs for a wide range of applications, including computational chemistry, sparse matrix solvers, sorting, searching, and physics models [1].

2.1 GPU vs CPU

A simple way to understand the difference between a GPU and a CPU is to compare how they process tasks. A CPU consists of a few cores optimized for sequential serial processing while a GPU has a massively parallel architecture consisting of thousands of smaller, more efficient cores designed for handling multiple tasks simultaneously [6].

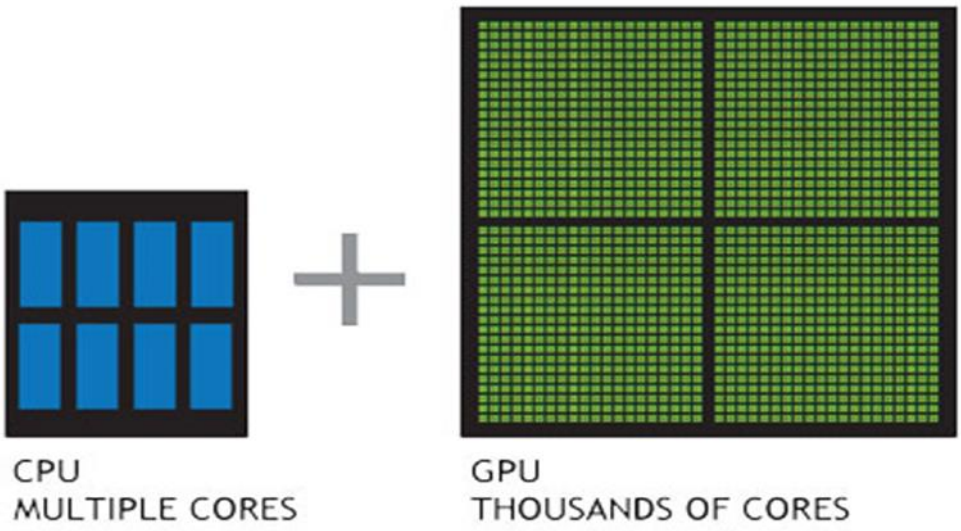

Figure 1:GPU and CPU core architecture [6]

The CUDA programming model is a heterogeneous model in which both the CPU and GPU are used. In CUDA, the host refers to the CPU and its memory, while the device refers to the GPU and its memory. Code run on the host can manage memory on both the host and device, and also launches kernels which are functions executed on the device. These kernels are executed by many GPU threads in parallel [7].

2.2 Threads and thread blocks

All threads which are called by the kernel run the same code. Each thread has an ID that it uses to compute memory addresses and make control decisions. CUDA threads are organized in blocks and can communicate within their own block. Blocks are organized into a one-dimensional, two-dimensional, or three-dimensional grid of thread blocks [8].

Developers can manipulate threads by using special built-in variables: ***threadIdx, blockIdx, blockDim, gridDim.*** Provided variables help developers to better understand the grid system and write more understandable code.

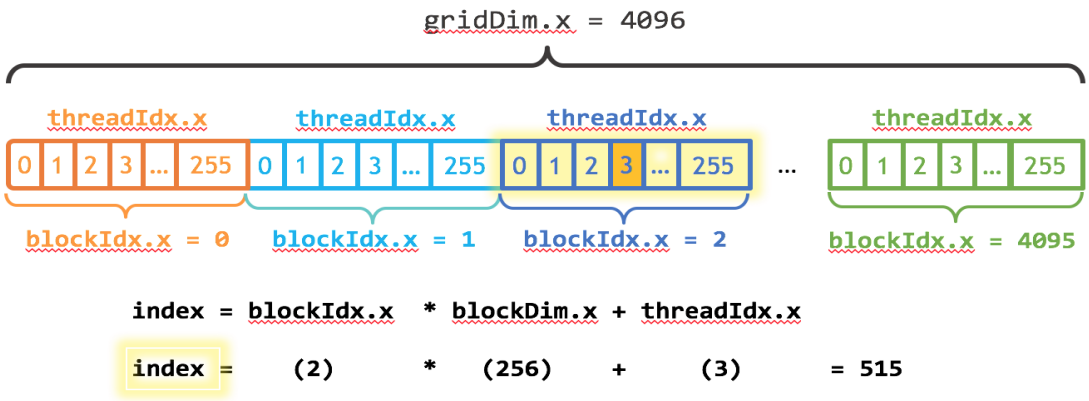

Figure 2: The CUDA parallel thread hierarchy [7]

2.3 Kernel execution

Let's return back to our example which was discussed in the introduction. We have two arrays A and B, and we what to perform multiplication operation. Result will have stored in the array named C. Resulting values will be located in the array according by their respected index. We will lock at the standard/sequential approach and parallel using a CUDA kernel.

As it was mentioned in the introduction, we only need a function that can loop through both arrays sequentially and the task is complete.

```
// Standard CPU sequential looping
void multiplyNumbers(int  * A, int  * B, int  * C) {
	for (int i = 0; i < 1000; i++) {
		C[i] = A[i] * B[i];
	}
}
```

CUDA solution is different but not difficult. CUDA kernel is defined by using the "__global__" declaration. This tells the compiler that this code will be performed on the GPU. We call the kernel from the CPU in main function. point. CUDA programs launch parallel kernels with the extended function-call syntax kernel

<<<dimGrid, dimBlock>>> (... parameter list ...);

where dimGrid and dimBlock are three-element vectors of type dim3 that specify the dimensions of the grid in blocks and the dimensions of the blocks in threads, respectively [1].

```
__global__ void multiplyNumbersOnCUDA(int  * A, int  * B, int  * C) {

	// create a global id for every used thread
	int id = blockIdx.x  *  blockDim.x + threadIdx.x;
	if (id < 1000) {		// perform parallel multiplication
		C[id] = A[id] * B[id];
	}
}

int main() {
	// Call device kernel from host
	multiplyNumbersOnCUDA <<<(1000 / 32) + 1, 32 >>> (A, B, C);
	cudaDeviceSynchronize();
	//...
}
```

Since we want to cover all the elements of the array we need to make sure that we use enough threads. For this example, we will use 32 threads for every block, thus we divide 1000 by 32 and get the number of blocks. When doing a division, we can get a non-integer so to avoid problems we increment resulting number by 1.

Each thread receives a global id index which depends on the multiplication of block dimension and block id with the addition of a thread index inside the block. This allows developers to track every used thread in the kernel.

“cudaDeviceSynchronize()” blocks host until the device has completed all preceding requested tasks and returns an error if one of the preceding tasks has failed [29].

2.4 Transparent scalability

The ability of kernels to transparently scale to different GPUs resolve from the hardware being able to schedule blocks to execute on parallel Streaming Multiprocessors (SM) where a multiprocessor is an array of processors that execute one or more blocks of threads.

Let’s imagine we have a CUDA program which will execute 8 blocks with 32 threads each. We want to test this code on 2 different GPUs. First GPU has 2 SM processors, second has 4 SM processors. During execution first GPU will schedule 4 blocks for every SM. In case of second, each SM will execute only 2 blocks. All this occurs without any modification to the source code. This gives an ability run CUDA code on all NVIDIA cards and the more SMs GPU has the better is the performance.

It is important to note that SMs instructions are issued per warp which consists of 32 threads each [30]. 32 is the minimum number that was chosen by NVIDIA developers

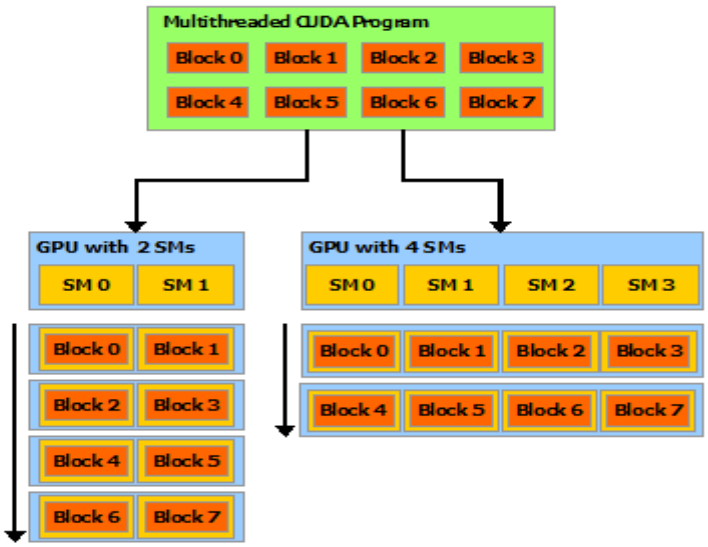

Figure 3:Automatic Scalability [8]

3 Databases and CUDA

Database with huge amount of data requires much time to process. It is time consuming to perform many operations on it, resulting in degradation of performance. CUDA is such a programming model by which performance in terms of time for any computation is improved [18].

3.1 In-memory databases

In-memory databases systems are database management systems where the data is stored entirely in main memory. In memory systems have been shown to be 50,000 times faster than disk based systems. This increase in speed along with a falling cost and non-volatility of computer memory has led to a greater interest in in-memory databases from the mid-2000's [15].

Current database management systems were designed assuming that data would reside on disk. However, memory prices continue to decline; over the last 30 years they have been dropping by a factor of 10 every 5 years. The latest Oracle Exadata X2-8 system ships with 2TB of main memory and it is likely that we will see commodity servers with multiple terabytes of main memory within a few years. On such systems the majority of OLTP databases will fit entirely in memory, and even the largest OLTP databases will keep the active working set in memory, leaving only cold, infrequently accessed data on external storage [16].

Both conventional disk management systems (DRDBMSs) and main memory database management systems (MMDBMSs) process data in main memory, and both keep a (backup) copy on disc. The key difference between them is that in an MMDBMS the primary copy of the database lives permanently in main memory. Even if the whole database of a DRDBMS is cached in main memory, it will not provide best performance since a DRDBMS is not tuned for this case. In MMDBMSs the situation is different: Since data is guaranteed to stay present in main memory, index structures and all the other parts of the database do not need to consider disk access and can be tuned for low computation cost [17].

When it comes to indexing in a disk based system is to locate a record on disk as quickly as possible based on an index key and record identifier. In an in-memory system index keys do not need to be stored. Indexes are implemented as record pointers which point to the corresponding record containing the key. In effect the record identifier is implemented as the record pointer. Because an index key is not stored in the index there is no duplication of key values in the index structure which reduces it size. Also pointers are all the same size so the need to manage variable length keys goes away making index implementation simpler [15].

In regards to query processing, a common clause in SQL is ORDER BY. In a disk based system this is fast if the data in the table being targeted by the query is stored in the order requested. Often, in fact usually, this is not the case. The next best option is to use an index scan. This carries the overhead of random access to records which may result in a high input/output cost. The problem is exasperated if sorting is required on multiple columns. In an in-memory database, as seen in the indexing, the only entries in the index are the record identifiers which are implemented as record pointers. In other words, index entries point directly to the memory address where data resides, so a random scan is no more expensive than a sequential scan, and of course there is no disk I/O. Buffer pool management is also unnecessary because you don't need to cache data in main memory because you are not doing any disk I/O [15].

3.2 Row-based and columnar-based databases

Since we will work with data which consists of millions values, it is wise to think about its structure. As was said in the introduction, we will have a two-dimensional array in which it is important to set its dimensions. It is possible to have 10 millions of arrays each consisting of 5 element array or vice versa. It is logical to qualify this two-dimensional array as a matrix which consists of rows and columns. So we can approach the data structure as row based or as columnar based. In general, deciding what database structure must be used is important topic of discussion. Each structure has its advantages and disadvantages.

3.2.1 Row-based

When developers talk about row-based approach it refers to traditional database management systems (RDBMS). RDBMS is the first database management system which we usually start to learn first. RDBMS is very useful and very straight forward way of storing data.

Sales			
Product	**Customer**	**Date**	**Sale**
Beer	Thomas	2011-11-25	2 GBP
Beer	Thomas	2011-11-25	2 GBP
Vodka	Thomas	2011-11-25	10 GBP
Whiskey	Christian	2011-11-25	5 GBP
Whiskey	Christian	2011-11-25	5 GBP
Vodka	Alexei	2011-11-25	10 GBP
Vodka	Alexei	2011-11-25	10 GBP

Figure 4: Row-based data [20]

When developers talk about row-based approach it refers to traditional database management systems (RDBMS). RDBMS is the first database management system which we usually start to learn first. RDBMS is very useful and very straight forward way of storing data.

Unfortunately, we start to experience difficulties when volume of data becomes really big. Here is the list of issues that come with this approach:

- **Indexing**

This is the most serious problem as due to indexing the size of the database increases. Also the process is slowed down because search time for the record becomes more in large table due to record by record search. The number of indexes that can be made is also limited. Retrieving and writing operations take much of the computational time because of moving entire record. So, all columns cannot be indexed and used effectively [19].

- **Optimization in the case of many indices**

A second problem is that the traditional database management system can't resolve a query and optimize the process to use only indexed fields if there are too many indices involved in the query. This leads to degradation of the system performance [19].

- **Fixed record structure**

The RDBMS has a fixed table structure. If new data or field is to be added, then table structure is to be modified or new field should be added at other place. It is very difficult to maintain the structure of table at all times [19].

- **Time to time reorganization of database**

Real data is all the while modified or changed in a period of time. This change should be incorporated in the database. Because of this many times the index is modified. This is very tedious job to deal with. Thus maintaining the database structure becomes more difficult [19].

3.2.2 Column-based

In columnar database the values stored in column one are stored in one set, all the values in column two in another set, and so on. In addition to the values, the information needed to reinsert them into the proper position in the original record format is stored with each set. From the simplicity of the columnar approach accrue many benefits, especially for those seeking a high performance environment to meet the growing needs of extremely large analytic databases. These key factors are seamlessly engineered into a column-oriented database, which enable reasonably-priced, benchmark-busting performance to meet an organization's business intelligence needs [19].

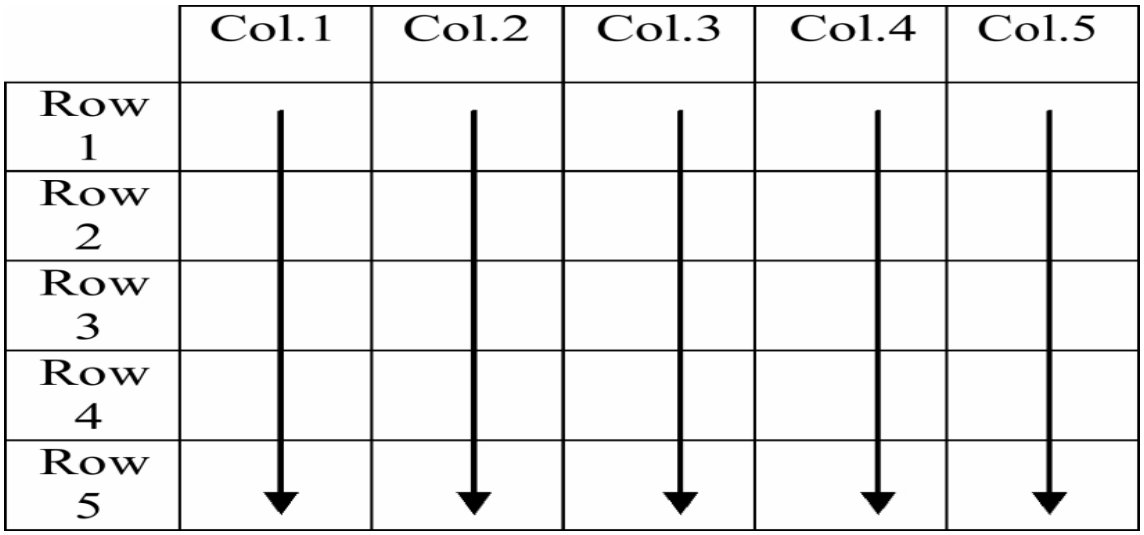

Figure 5: Column-based structure [19]

The columnar database plays important role in data warehouse systems but when insertion, modification, deletion operations are considered the performance of columnar database becomes poor. The insertion of record at right place becomes costly in this case as finding the right position for insertion is difficult. Along with this, the performance of column-oriented database system degrades in the case of importing, exporting, tuple construction and bulk reporting [19].

3.3 Overview of CUDA for databases

A database system performs a significant amount of repeated calculations on different data. This can occur in table joins or in conditional statements. Both of these database system functions can require significant computation time, slowing system performance and providing an opportunity for GPU programming to offer significant advantages [22].

3.3.1 Query speedups

Before creating databases that run only on GPU, researchers and developers started experiment with SQL queries. The goal was to see how fast, for example, SELECT WHERE queries can work on a GPU and also try adding JOIN operations. For example, in 2009, there was an attempt to make a SQL accelerator which proved that SELECT WHERE query is much faster. In regards to SELECT JOIN a speed up was also detected.

Although, this has required an allocation of significantly big chunks of memory, since JOIN operation is a Cartesian product – result of merging two tables [21].

A similar experiment was done in 2010 in the University of Virginia, where they focused on attempting to accelerating SELECT queries using CUDA with a SQLite – an open source database which can be configured to work in main memory. SQLite was attractive primarily for its simplicity, having been developed from the ground up to be as simple and compact as possible. The source code is very readable, written in a clean style and commented heavily. The data used for performance testing has five million rows with an id column, three integer columns, and three floating point columns [3].

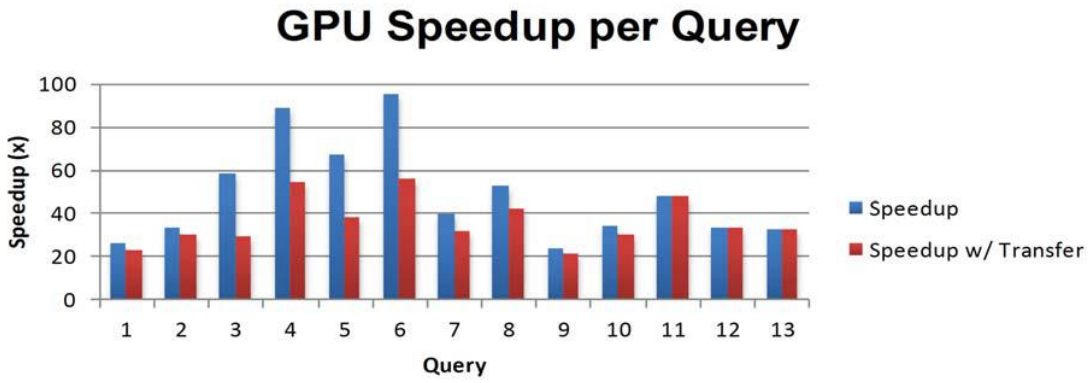

Figure 6: GPU Speedup per Query [3]

This project simultaneously demonstrates the power of using a generic interface to drive GPU data processing and provides further evidence of the effectiveness of accelerating database operations by offloading queries to a GPU. Though only a subset of all possible SQL queries can be used, the results are promising and there is reason to believe that a full implementation of all possible SELECT queries would achieve similar results [3].

3.3.2 Database projects and extensions

Constant working with CUDA allowed to create different extensions and one of them is PG-Storm. PG-Strom is an extension designed for PostgreSQL v9.5 or later, to off-load a part of CPU intensive workloads to GPU (Graphic Processor Unit) devices, and execute them in parallel asynchronously. This module is designed to reduce response time of complicated queries executed on large data set (like, data analytics or batch processing), on the other hands, it is not preferable to run transactional workloads or heavy concurrent

processing. It allows to perform full table scan, tables join, group by/aggregation and projection [24].

As time went on, more experiments took place. There were attempts to use CUDA with MapReduce[22] – programming model and an associated implementation for processing and generating large datasets that is amenable to a broad variety of real-world tasks[23]. This also was done to test searching operations on index file in database query processing.

The time of execution of searching operation of index file shows that a GPU can be used to accelerate SQL databases whereas Map Reduce approach also takes less execution time for searching index file in Hadoop Environment. In this case it is possible to achieve better acceleration because very often more operations can be run independently than in a single SQL operation. It depends on the different versions of GPU card which can increase processing speed in terms of data retrieval from database. Future work in this system will require expanding the database implementation to tile databases that are too large for the system into the GPU memory. Even with the advent of direct GPU memory access in CUDA 6, direct management of the GPU memory spaces will yield much better performance results [22].

Since 2012, MapD was developed by gg. MapD Core is an in-memory, column store, SQL relational database that was designed from the ground up to run on GPUs. The parallel processing power of GPU hardware enables the MapD Core database to query billions of rows in milliseconds using standard SQL [26].

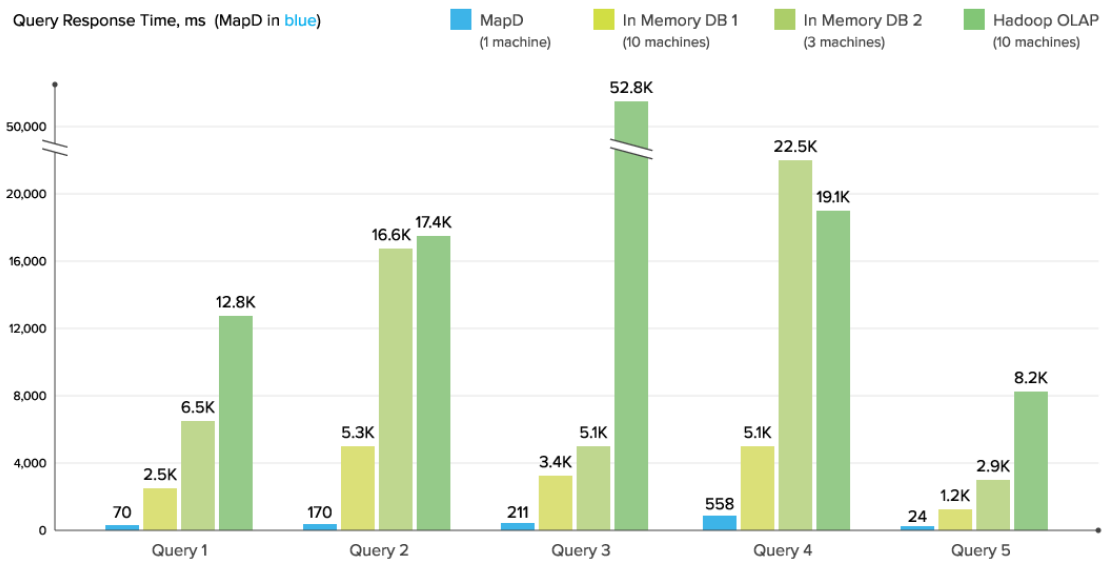

Figure 7 Query response time in MapD [28]

In 2016, BlazingDB introduced its database. BlazingDB started as a simple SQL database that helped handle large data sets on consulting projects. Eventually it was upgraded to become an elastic SQL data warehouse. BlazingDb thanks to computing power of the GPU, can scan millions and millions of rows in seconds [25].

4 Goals and scenarios for experiments

Before performing any experiments, we need to define the task and understand CUDA memory architecture. CUDA has access to different types of memory on the GPU will be discussed in fort coming subchapters.

4.1 High level view of what we need

In order to measure CUDA performance we will define a two-dimensional array which will be regarded as a “matrix”. The matrix will consist of 10 million records and 5 fields maximum. Matrix dimensions are defined by size parameters – R and C. For simplicity, stored data will be represented as integer values. Data range in matrix will be from 1 to 10.

After matrix definition and data insertion we will apply a search method that will count all the records which contain value of 10. This operation will be performed in RAM memory and in GPU memory using CUDA.

Since memory transfer is an essential part of CUDA programing it is wise to apply search function more than once. Before applying search function, a second time we will update the data by replacing value 7 with 10 receiving a doubled result.

Search and update functions execution will be measured in real time. CPU and GPU results will be stored in tables and compared to each other.

4.2 Explain CUDA memory structure

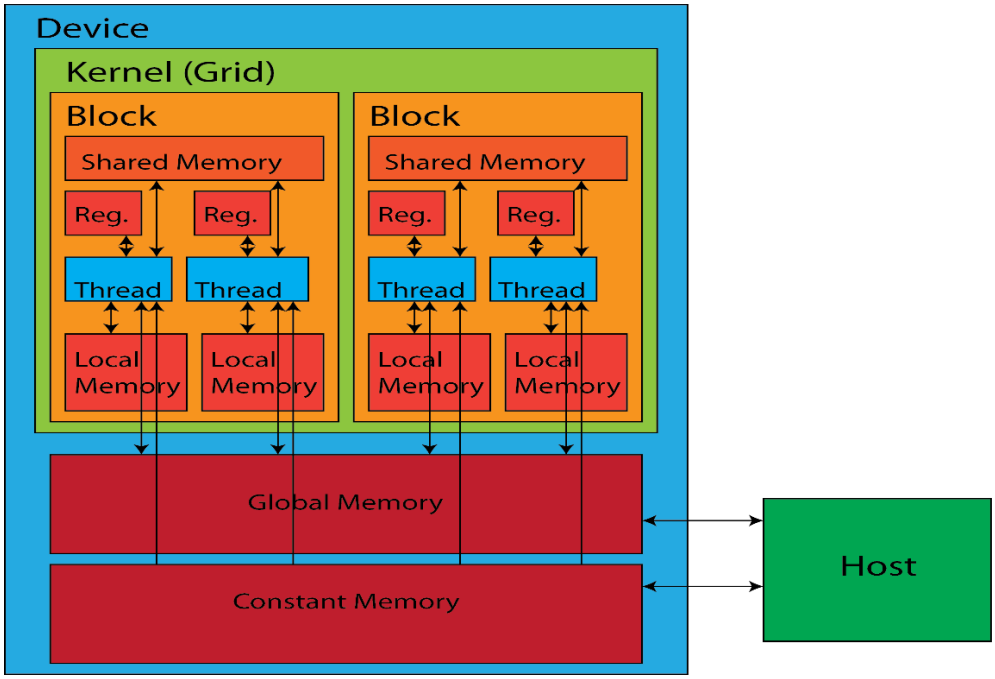

Figure 8: CUDA memory access [10]

Since memory transfer is an essential part of CUDA programing it is wise to apply search function more than once. Before applying search function, a second time we will update the data by replacing value 7 with 10 receiving a doubled result.

Global memory takes the majority of memory on the GPU device, stored in off-chip DRAM, and with the slowest latency on board [12]. In order to work with global memory, we use a set of defined functions:

- cudaMalloc()
- cudaMallocManaged()
- cudaMemcpy()
- cudaFree()

Developers can only use these functions from the CPU, because the GPU is a slave device that receives FDTD task information and data from host computer via a Peripheral Component Interconnect Express (PCIe) bus [13]. When creating code for kernel or

kernel execution, developers must use declarations such as "__global__" and "__device__" which also work on global memory.

By default, register memory or file (another name) takes all CUDA variables that are created inside GPU functions. The number or registers is calculated during compilation process. If there are not enough registers variables are buffered to local memory. This process is also known as "spilling". Local memory accesses have same high latency and low bandwidth as global memory accesses and are subject to the same requirements for memory coalescing [8]. Local memory can be regarded as SWAP on the CPU.

The fastest memory is shared memory because it is on chip [8]. Shared memory accessible directly by threads and programmer managed. This hierarchy enables a programmer to control data flow and minimize access latency [12]. In order to work with shared memory directly and assigned threads in CUDA code we use "__shared__" declaration.

Constant memory can be used to declare constant variables and can also allocate memory for array of elements. However, it is not possible to do dynamic allocation of constant memory. Dynamic allocation can be done in global memory using malloc and copy functions listed above.

Type	Scope	Location	Latency	Bandwidth	Access
Global	Grid	DRAM	400-600 clock	100GB/s	Read/Write
Local	Thread	DRAM	400-600 clock	100GB/s	Read/Write
Register	Thread	On chip	Immediate	Immediate	Read/Write
Shared	Block	On chip	4-6 clock	200GB/s	Read/Write
Constant	Grid	Cache or DRAM	4-600 clock	200-300GB/s	Read only
Texture	Grid	Cache or DRAM	4-600 clock	200-300GB/s	Read only

Figure 9 : GPU memory structure

4.3 Standard and unified memory

The fastest memory is shared memory because it is on chip [8]. Shared memory accessible directly by threads and programmer managed. This hierarchy enables a programmer to control data flow and minimize access latency [12]. In order to work with shared memory directly and assigned threads in CUDA code we use "__shared__" declaration.

Memory management is crucial part when attempting to transfer data from CPU to GPU. Initial data must be declared on the host. After data declaration it is necessary to declare a set of pointers for host data and a separate set of pointers for data on the device. By manipulating host and device pointers we allocate required space on the GPU and copy data from host to device. Every step of allocation and transfer must be verified since there is no guarantee that it will always be successful. If written code did not highlight any errors, then we can execute the kernel. After successful execution which must be verified as well we copy data back to host variables in order to continue working with it or display it on screen.

The common programming model is as follows [12]:

1. CPU serial code and data initialization
2. Data transfer to the GPU
3. Parallel computation execution
4. Data transfer back to the CPU.
5. CPU serial code

Because of the complexity of data management NVIDIA started to look for a way to make this process much easier for developers. And since CUDA 6.0 Unified Memory Access (UMA) was introduced.

Unified Memory creates a pool of managed memory that is shared between the CPU and GPU, bridging the CPU-GPU divide. Managed memory is accessible to both the CPU and GPU using a single pointer. The key is that the system automatically migrates data allocated in Unified Memory between host and device so that it looks like CPU memory to code running on the CPU, and like GPU memory to code running on the GPU [11].

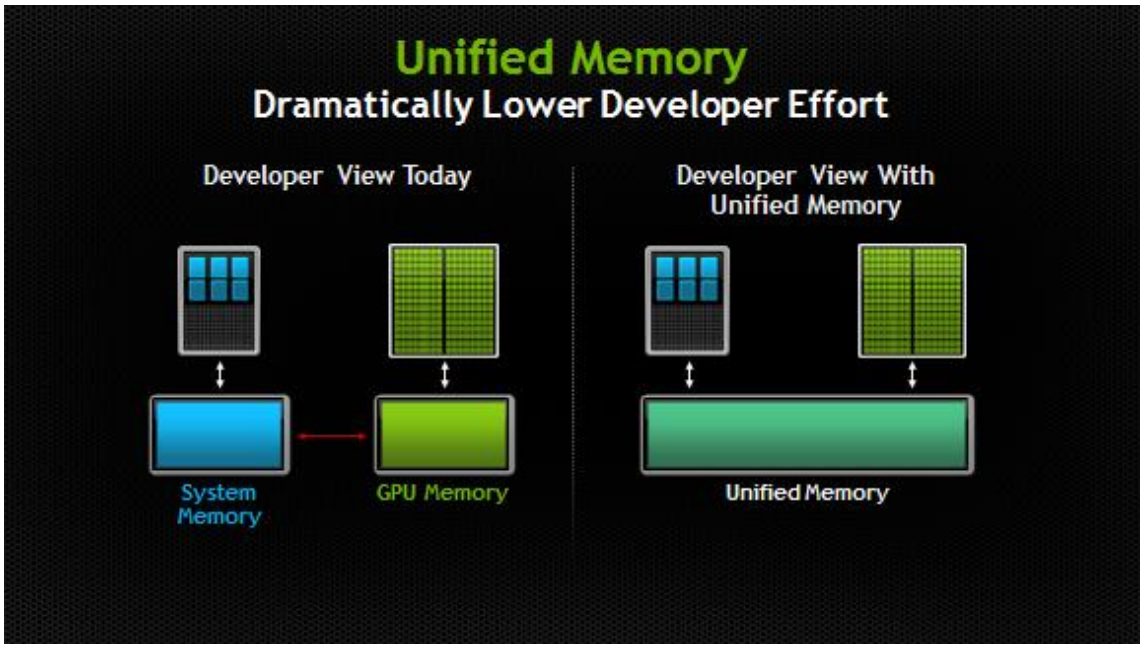

Figure 10: Introduction to Unified Memory [11]

Programming model with UM [12]:

1. CPU serial code and data initialization
2. Parallel kernel execution
3. Synchronization between CPU and GPU
4. CPU serial code

Further research into advantages and disadvantages of using unified memory to standard approach will be demonstrated in the next chapter.

5 Experiments with CUDA

In this chapter we perform all the necessary experiments to determine how efficient is GPU with CUDA vs CPU with RAM. Here are displayed all the experiments and their results.

All the experiments are done on my home PC which has these specifications:

Operation system: UBUNTU 16.04

Processor: Inrtel® Core(TM) i7-6700K CPU @ 4.00GHz (8 CPUs) 4.01 GHZ

RAM: 8 GB

System type: 64 bit

Video card: GeForce GTX 970 4GB

CPU test files were written using **NetBeans IDE**. CUDA test files were written using **Nsign Eclipse IDE** provided by the **CUDA-8.0 toolkit**.

5.1 Experiments structure

In this chapter we perform all the necessary experiments to determine how efficient is GPU with CUDA vs CPU with RAM. Here we have all experiments displayed with their results.

The goal of the experiments to prove that CUDA can improve search capabilities on a two-dimensional array with dimensions of 50 million elements. Experiments will be performed in two types of memory. First in main memory or RAM, second on GPU memory using CUDA.

Since we regard our matrix as a table we can approach its structure as columnar based or as row based. In theory which was establish in previous chapter columnar based representation must perform faster vs row based structure. Both structures will be tested on the CPU and on the GPU.

After measuring the performance an identical experiment will be performed in CUDA. Based on the established theory in previous chapters, performance in CUDA should produce faster results. In addition to that, it is important to measure how time consuming is the memory managing process which is an integral part of the CUDA programming model.

Finally, when experimenting with CUDA it is necessary to verify CUDA performance with unified memory. How does really unified memory improve the code performance and code complexity?

Test scenarios:

- Row approach in RAM memory
- Columnar approach in RAM memory
- Row approach in CUDA **without** unified memory
- Columnar approach in CUDA **without** unified memory
- Columnar approach in CUDA **with** unified memory
- Columnar approach in CUDA **with** unified memory

For each scenario we have a working project – 6 in total. Every project consists of 3 files. Measurements and functions call are in the main file. Functionality for methods is in the second source file. Finally, all the constants and function declarations are in the header file.

In this thesis results for every project will be demonstrated. However, in regards to code description, it was decided not to go through every project. Code description will be provided only for 3 projects total, since we want to highlight the usage of functionality provided by CUDA.

First will be about using a standard CPU with row-based structure. Second will be on GPU using row-based structure without unified memory. The last will be also on GPU with same structure but with unified memory.

It is important to mention that every operation that the code does is measured in real time. Every important method is located between C library function “clock(void)” which returns the number of clock ticks elapsed since the start of the program in t_size format. This will not be demonstrated in code description since we concentrate on algorithms at this moment. Results of time measurements will be demonstrated in subchapters related to results and analysis. This can be seen in the source files.

All created code for this thesis will be uploaded to github. Links will be provided.

5.2 Structure of test code for standard CPU

Project name: TestRowBasedHost

First, we declare a two-dimension array using the “new” operator where R = 10 million and C =5. In order to refer to the array, we will use a pointer to first 5 variable row (*tableRow)[5].

```
// We create row based table. It will have 10 million rows and 5 columns
    int (*tableRow)[C] = new int[R][C];
```

Second, we will use this pointer as an argument for a “fillField” function. It will loop through every row by incrementing the pointer, adding data from 1 to 10 to specifically targeted field. In our experiment, we chose number 2.

```
/* Every row will have its second field with data
 fillColumn will add numbers from 1 to 10
 */
void fillField(int (*tableRow)[C]) {

    int i = 0;
    int current = StartValue;
    int *row;

    while (i < R) {
        row = (int *) tableRow;
        tableRow++;
        if (current > EndValue)
            current = StartValue;
        row[TargetField] = current;
        i++;
        current++;
    }
}
```

Third, we scan our filled table and try to count all rows which have index number 2 equal to 10. Function returns total number of counted values which is 1 million.

```
// Function will find specific field in a row
int scaleRows(int (*tableRow)[C]) {

    int count = 0;
    int *row;

    for (int i = 0; i < R; i++) {
        row = (int *) tableRow;
        if (row[TargetField] == SearchTarget)
            count += 1;
         tableRow++;
    }

    return count;
}
```

After we successfully replace (re factor) array data. We scan the matrix once more. Since we have 10 million records and data is in range from 1 to 10 first scan result must be 1 million values scanned. After re factor final result is 2 million.

5.3 Present the results of test code for CPU

Data structure: Row based

Command: $ g++ -o3 -o row TestRowBasedHost.cpp RowBased.cpp

First search: 1000000 fields with target value 10

Second search: 2000000 fields with target value 10

Total execution

Operation	Time Result
Fill data	30,7 ms
First search	18,1 ms
Re-fill function	18 ms
Second search	18,7 ms
Total Result	***Ca 85,5 ms***

Table 2: Total execution time for row-based structure on CPU

Data structure: Column based

Command: $ g++ -o2 -o column TestColumnBasedHost.cpp ColumnBased.cpp

First search: 1000000 fields with target value 10

Second search: 2000000 fields with target value 10

Total execution

Operation	Time Result
Fill data	21,4 ms
First search	17,4 ms
Re-fill function	17,3 ms
Second search	16,7 ms
Total Result	***Ca 72,8 ms***

Table 3:Total execution time for column-based structure on CPU

5.4 Explain the structure of test code for CUDA

In case of CUDA experiment we will use 2 projects – “TestRowBasedDevice” and “TestRowBasedDeviceUnif”. First uses standard memory. Second – unified memory framework.

5.4.1 Standard memory framework

Matrix initialization and data filing is identical to a CPU program. After this step, we start working with CUDA functions.

First we need to create a set of pointers. First set will be used for data used on the host and second set is for the device.

First we need to start CUDA initialization. This can be done with calling cudaFree() function.

```
// We create row based table. It will have 10 million rows and 5 fields
int(*h_tableRow)[C] = new int[R][C];

int count = 0;

int *h_count = &count; // initialize host variable for result

int(*d_tableRow)[C]; // initialize cuda pointer for every row
int *d_count; // initialize cuda variable for result
```

Second, we need to initialize CUDA.

```
//*************** Initialize cuda *******

cudaFree(0);
```

Third, we allocate space for matrix on the device. Also need to allocate space for a result counter. Since there is no guarantee that CUDA function will work appropriately, we add if blocks for verification.

```
//**************** Copy cuda memory to device **********

if (cudaMemcpy(d_tableRow, h_tableRow, R * C * sizeof(int),
       cudaMemcpyHostToDevice) != cudaSuccess) {
       printf("\nFailed to copy d_tableRow to device");
       return 0;
}

if (cudaMemcpy(d_count, h_count, sizeof(int), cudaMemcpyHostToDevice)
       != cudaSuccess) {
       printf("\nFailed to copy d_count to device");
       return 0;
}
```

Fifth, after all the data copied we can launch the kernel execution. Number of block will be determining by division of number of rows, which is 10 million by number of threads which is 1024 – max threads per block.

```
scaleCudaRow <<<R / 1024 + 1, 1024 >>>(d_tableRow, d_count);
cudaDeviceSynchronize();
```

Functionality of the kernel is not as easy as in the standard CPU example. The difficulty comes when you have to do incrementing. We encounter a problem of race condition which is a common problem in multithreaded applications. It is necessary to make the process atomic – without any threads interference [14]. For this we have atomicAdd.

Example:

int `atomicAdd`(int* `address`, int `val`);

This atomicAdd function can be called within a kernel. When a thread executes this operation, a memory address is read, has the value of ‘val’ added to it, and the result is written back to memory [14].

Although, just adding atomicAdd() is not enough. When we do an adding process inside a kernel. atomicAdd will provide us with a full result only after kernel will finish executing. However, in our test scenario it is required to move through every row by constantly incrementing the matrix pointer. To solve this challenge, it was decided to declare a global “__device__” variable “incrementor” which is initially equals to 0.

```
// Do a search in matrix using cuda
__global__
void scaleCudaRow(int(*d_tableRow)[C], int *d_count) {
	int id = blockIdx.x * blockDim.x + threadIdx.x;

	if (id < R) {
		d_tableRow += atomicAdd(&incrementor, 1);
		int *row = (int *)d_tableRow;
		if (row[TargetField] == SearchTarget) {
			atomicAdd(d_count, 1);
		}
	}
}
```

Because we will continue to work with rows with CUDA, it is necessary to set "incrementor" back to 0.

```
resetIncrementor <<< 1, 1 >>> ();
cudaDeviceSynchronize();

__global__  void  resetIncrementor() {
	incrementor = 0;

}
```

After completing first search, we return result variable "d_count" back from device to host. Result will be set in main memory and pointer "h_count" will be used for access. The result will be 1 million found rows.

```
if (cudaMemcpy(h_count, d_count, sizeof(int), cudaMemcpyDeviceToHost) !=
cudaSuccess) {
	printf("\nFailed to copy d_count to host");
	cudaFree(d_tableRow);
	cudaFree(d_count);
	return  0;

}
```

Having stored the first search result, we need to substitute matrix data. This can also be done in CUDA.

```
substituteFieldCudaValue <<<R / 1024 + 1, 1024 >>> (d_tableRow);
cudaDeviceSynchronize();


__global__  void  substituteFieldCudaValue(int(*d_tableRow)[C]) {
	int id = blockIdx.x  *  blockDim.x + threadIdx.x;
	if (id < R) {
		d_tableRow += atomicAdd(&incrementor, 1);
		int  * row = (int  *)d_tableRow;
		if (row[TargetField] == SubstituteValue) {
			row[TargetField] = SearchTarget;

		}

	}

}
```

Finally, after setting "h_count" to 0 and copying it back to device we launch search kernel for the last time. After doing same copy operation from device to host we will have our result who's values equals to 2 million found rows.

5.4.2 Unified memory framework

When using unified memory framework code designed becomes more easy. We do not need separate pointer for the device and we do not need to copy data to device and back to host. Unified memory gives us a chance to use only one set of pointers.

We will only use two pointers.

```
int(*d_tableRow)[C]; // declare pointer for every row
int *d_count; // declare pointer to variable that stores counting result
```

We initialize our pointers in unified memory.

```
//*****************Malloc data in
unifiedcudamemory*****************************

cudaMallocManaged(&d_tableRow, R * C * sizeof(int));
cudaMallocManaged(&d_count, sizeof(int));
```

Then we take the same steps as we took in the previous experiment only without the copying.

We set counter pointer to zero and need to initialize CUDA.

```
*d_count = 0;

//*************** Initialize cuda *******

cudaFree(0);
```

After that we use the same function from filling data to kernel execution with data substitution.

```
fillField(d_tableRow);

scaleCudaRows <<<R / 1024 + 1, 1024 >>>(d_tableRow, d_count);
cudaDeviceSynchronize();

resetIncrementor <<<1, 1 >>>();
cudaDeviceSynchronize();

substituteFieldCudaValue <<<R / 1024 + 1, 1024 >>>(d_tableRow);
cudaDeviceSynchronize();

resetIncrementor <<<1, 1 >>>();
cudaDeviceSynchronize();

scaleCudaRows <<<R / 1024 + 1, 1024 >>>(d_tableRow, d_count);
cudaDeviceSynchronize();
```

The result is the same but code design became more understandable and compact.

5.5 Present the results of test code for CUDA

Before we demonstrate the result it necessary to mention that some operations worked faster than 1 ms. They did not have any impact on total execution so they were left out from the result tables. It is possible to see all the results for all the projects; this can be done by compiling the source files. All information will be displayed in the command prompt.

Data structure: Row based

Memory framework: standard memory framework

First search: 1000000 fields with target value 10

Second search: 2000000 fields with target value 10

Command: $ /usr/local/cuda-8.0/bin/nvcc - row RowBasedDevice.cu RowBased.cu

Total execution

Operation	Time Result
Fill rows	62,2 ms
Initialize CUDA	71,8 ms
CUDA Malloc column to device	0,3 ms
CUDA Malloc result variable to device	0,2 ms
Copy rows info to device	23,1 ms
First cuda search	**1,342 ms**
Cuda Re factor function	**1,7 ms**
Second cuda search	**1,279 ms**
Total Result	***Ca 162,1 ms***

Table 4:Total execution time for row-based structure on GPU

Data structure: Column based

Memory framework: standard memory framework

First search: 1000000 fields with target value 10

Second search: 2000000 fields with target value 10

Command:

$ /usr/local/cuda-8.0/bin/nvcc - column ColumnBasedDevice.cu ColumnBased.cu

Total execution

Operation	Time Result
Fill rows	19,4 ms
Initialize CUDA	84,3 ms
Cuda Malloc column to device	0,2 ms
Cuda Malloc result variable to device	0,1 ms

Copy rows info to device	5,1 ms
First cuda search	**0,4 ms**
Second cuda search	**0,4 ms**
Total Result	***Ca 110,5 ms***

Table 5: Total execution time for column-based structure on GPU

Data structure: Row based

Memory framework: unified memory framework

First search: 1000000 fields with target value 10

Second search: 2000000 fields with target value 10

Command:

$ /usr/local/cuda-8.0/bin/nvcc - column RowBasedDeviceUnif.cu RowBased.cu

Total execution

Operation	Time Result
Initialize CUDA	2,2 ms
Cuda Malloc column to device	108,8 ms
Cuda Malloc result variable to device	124,2 ms
Fill column	60,9 ms
First cuda search	**17,3 ms**
Cuda Re factor function	**1,8 ms**
Second cuda search	**1,5 ms**
Total Result	***Ca 317 ms***

Table 6: Total execution time for row-based structure on GPU with unified memory

Data structure: Column based

Memory framework: unified memory framework

First search: 1000000 fields with target value 10

Second search: 2000000 fields with target value 10

Command:

$/usr/local/cuda-8.0/bin/nvcc –column ColumnBasedDeviceUnif.cu ColumnBased.cu

Total execution

Operation	**Time Result**
Initialize CUDA	1,9 ms
Cuda Malloc column to device	97,6 ms
Cuda Malloc result variable to device	113 ms
Fill column	25,4 ms
First cuda search	**3,7 ms**
Cuda Re factor function	**0,5 ms**
Second cuda search	**0,4 ms**
Total Result	***Ca 243 ms***

Table 7: Total execution time for column-based structure on GPU with unified memory

6 Experiment with CUDA and WhiteDB

WhiteDB is a lightweight NoSQL database library written in C, operating fully in main memory. There is no server process. Data is read and written directly from/to shared memory, no sockets are used between WhiteDB and the application program [31].

Our goal is to use WhiteDb on the GPU with CUDA instead of using it in shared or RAM memory and measure the performance. The idea is to reproduce the same experiment which was done in previous chapter. We need to create 10 million of rows of data. In each row we will have one field with a value from 1 to 10. WhiteDB will be instructed to count all the rows which have a value of 10.

WhiteDB has complex functionality. It is a very time consuming task to properly integrate CUDA with WhiteDB. This requires deep understanding of in memory databases and shared memory properties. This can be master's thesis in itself. The solution which was attempted in this thesis is to migrate WhiteDB code and to adapt to CUDA rules.

Since WhiteDB can work in main memory and on local/disc memory, it was decided to try and reproduce local memory database initialization on CUDA. Initialization is done with function:

```
void* wg_attach_local_database(int size)
```

Returns a pointer to local memory database, NULL if failure. Size is given in bytes. The database is allocated in the private memory of the process and will neither be readable to other processes nor persist when the process closes. In every other aspect the database behaves similarly to a shared memory database.

In order reproduce this, I wrote a separate CUDA function which uses the same functionality but adapted to CUDA. This is not a difficult step, since all defined structures and macros functions can be used in CUDA.

```
__global__   void  initialize_db_on_cuda(void  * d_db, gint  * d_size) {
	if (threadIdx.x == 0) {
		d_db = wg_attach_local_cuda_database(*d_size);
	}
}
```

Compilation process was successful and during execution there was no null value returned. The end result was not guaranteed. In order to prove that database was properly initialized it was necessary to try and add integer values.

Value insertion process is complex. It uses operations, definitions and macros from different files. All function calls very traced and the final result was 10 CUDA files. All the functionality needed to have __device__ declaration.

Since we had multiple CUDA files, it is important to understand separate compilation in CUDA.

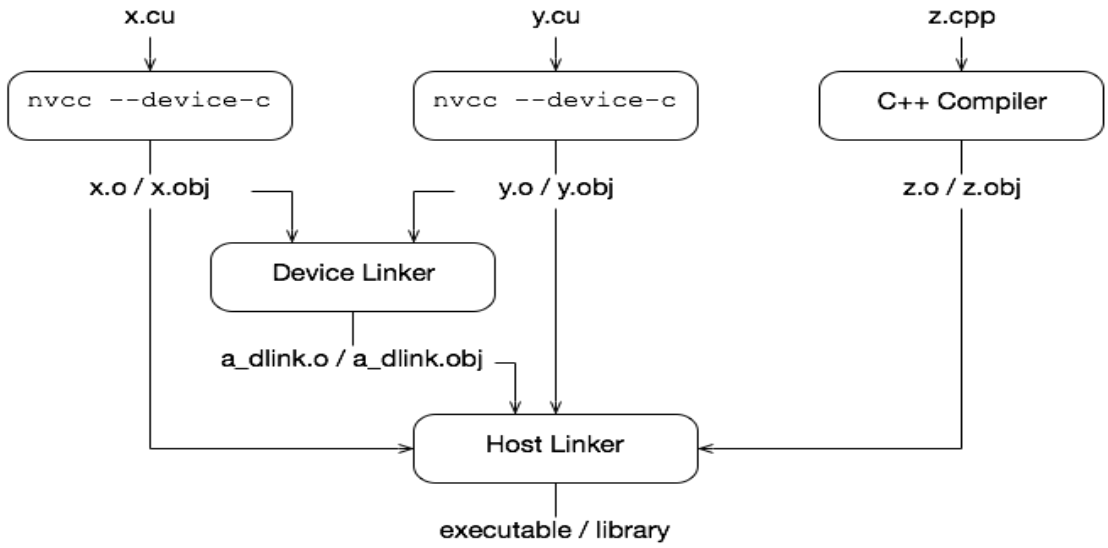

Figure 11: CUDA Separate Compilation Trajectory [27]

Compilation requires special flags.

Commands:

/usr/local/cuda-8.0/bin/nvcc –dc main.cu ../../../C/cualloc.cu ../../../C/cumem.cu ../../../C/cumem.cu ../../../C/culock.cu ../../../C/cudata.cu ../../../C/cuindex.cu ../../../C/cuquery.cu ../../../C/cuindex.cu ../../../C/cuhash.cu ../../../C/cucompare.cu

Created object files are required to be linked in order to make an executable.

/usr/local/cuda-8.0/bin/nvcc –rdc=true main.o cualloc.o cumem.o culock.o cudata.o cuindex.o cuquery.o cuhash.o custring.o cucompare.o -o executeMyCudaFile

./executeMyCudaFile

First execution process failed because CUDA does not know some standard C functions from standard C library – strlen, strcmp, memcpy.

To fix by creating a separate file custring.cu where code for these functions was written using open source sources.

```
#include < string.h >

__device__ size_t strlen(const char * str) {
	const char * s;
	for (s = str; *s; ++s)
		;
	return (s - str);
}

__device__ int strcmp(const char * s1,
	const char * s2) {
	while (*s1 == *s2++)
		if (*s1++ == 0)		return (0);
	return (*(unsigned char *)s1 - *(unsigned char *) --s2);
}

__device__ int memcmp(const void * s1,
	const void * s2, size_t n) {
	if (n != 0) {
		const unsigned char * p1 = (const unsigned char *)s1,
			*p2 = (const unsigned char *)s2;
		do {
			if (*p1++ != *p2++)
				return (*--p1 - *--p2);

		} while (--n != 0);

	}
	return (0);

}
```

After adding this file to the command line compilation was successful.

Command:

/usr/local/cuda-8.0/bin/nvcc –dc main.cu ../../../C/cualloc.cu ../../../C/cumem.cu ../../../C/culock.cu ../../../C/cudata.cu ../../../C/cuindex.cu ../../../C/cuquery.cu ../../../C/cuindex.cu ../../../C/cuhash.cu ../../../C/custring.cu ../../../C/cucompare.cu

Unfortunately, despite successful compilation, execution of result file did not give any results. Compiler did not catch and print out any errors. Program just successfully stops executing. By adding additional error checking functions or debugging also does not provide any information why program just stops and exit the process.

7 Comparative analysis

Provided results were very interesting. Judging by theoretical data, programs with CUDA should have improved the execution time. Instead we see that CPU programs (row and columnar based) complete their task faster. To understand the advantages and disadvantages we need to analyze the results more tediously.

7.1 Review search speed

Data structure	First Search	Re factor	Second Search
CPU Row Based	18,1 ms	18 ms	18,7 ms
CPU Column Based	21,4 ms	17,4 ms	17,3 ms
CUDA Row Based	1,3 ms	1,7 ms	1,3 ms
CUDA Column Based	0,4 ms	1,7 ms	0,4 ms
CUDA with Unified memory Row Based	17,3 ms	1,8 ms	1,5 ms
CUDA with Unified memory Column Based	3,7 ms	0,5 ms	0,4 ms

Table 8: Search speed results

By comparing search and re factor result of CPU and GPU we can confirm theoretical statements which for highlighted before. GPU with CUDA model managed to perform better that its CPU counterpart.

Column based structure has advantages on the CPU and on the GPU. GPU results demonstrate the difference better. The fact that the difference is not that significant mostly

depends on the fact that used data was not very big, although we had 50 million fields. It is likely that the program on both devices took more time to initialization and memory allocation so that it could not feel the data structure advantages and disadvantages. Also optimization flags for the C++ compiler on the CPU did not boost the performance as well. For future analysis it is required to use high volumes of data from 100 of millions to billions. For these experiments more advanced software and hardware will be needed. This volume of data is mostly used in modern companies which are used for machine learning, fraud detection and other data mining operations.

7.2 Review memory allocation speed

Data structure	CUDA initialization	First malloc	Second Malloc
CUDA Standard Row Based	71,8 ms	0,3 ms	0,2 ms
CUDA Standard Column Based	84,3 ms	0.2 ms	0,1 ms
CUDA with Unified memory Row Based	2,2 ms	108,8 ms	124,2 ms
CUDA with Unified memory Column Based	1,9	97,6	113 ms

Table 9: Memory allocations speed result

While working with CUDA an interesting pattern was noticed. Every first “malloc” function without using unified memory framework had a large time consuming spike. Second malloc operation did not have that issue. “Malloc” operations in C language have been usually fast, since we just allocate free space.

In attempt to find a solution, it was discovered that it does not matter what first CUDA function is used. Every first CUDA “call” will have this time spike. It was decided later to use “cudaFree(0)” as the first CUDA function. This call is regarded in this thesis as “CUDA Initialization”.

“cudaFee(void** devPtr)” is a function that frees memory in CUDA, it is similar to “free()” in C/C++.

Unfortunately, this initialization process consumes all the advantages in mine experiment which is provided by CUDA in other operations. However, the situation is different when we use unified memory.

Unified memory uses a new introduced cudaMallocManaged() function. As we can see in the results, first CUDA function takes maximum up to 2 ms. But memory allocation is a 100 times higher and it is shared between all allocation operations. In addition to this, row based first search is 15 times longer - 17,3 ms. As a result, unified memory framework made the execution time even longer.

This is an interesting predicament. On the one hand unified memory usage allowed to drastically reduce to code volume and made the designed flow a lot easier, as it was intended by the NVIDIA developers. On the other, on simple operation memory allocation process takes significantly more time and in advanced programs can become an issue.

In addition to that, the time spike can become on any allocation process. For example, second allocation is performed only for the result variable, which is only one integer and it takes more time than for a two-dimensional array of 50 million fields.

The provided result demonstrates that CUDA is better to be used with high volume data. It is more meant for difficult calculation which are process through multiple loops which can be process for hour on the CPU based platforms. With the understanding of the CUDA basics, it is necessary to try integrate CUDA with BIG DATA applications, picking specific time consuming processes.

8 Conclusion

This was our first interaction with CUDA programming model. It was an interesting opportunity to write code for a GPU than for CPU. By writing separate programs for CPU and GPU we managed to prove to ourselves that GPUs using CUDA can boost performance for specific of software's functionality. We also attempted to integrate CUDA with WhiteDB. Unfortunately, despite successful compilation the execution could not provide us any specific results.

In case of our thesis we managed to identify and measure the performance boost. However, all advantages that we achieved had been overshadowed by what we named as CUDA initialization process. This obstacle was an unexpected surprise. In addition to that, there is not much information about how to deal with this issue.

I believe that the lack of information is because CUDA is used for more complex tasks with large datasets. In these scenarios, such spikes are not an issue, especially if are dealing with BIG DATA that often takes hours to process. Unfortunately, the lack of time and experience did not give us a chance to start with large data volumes. That being said, it was not possible to quickly take the optimal path when you do not know the basics of GPU programming.

In regards, to WhiteDB, an attempt to compile an entire library was the first step but not the best one. When integrating CUDA with other software it is important to fully understand the algorithm, in our case WhiteDB. This would have allowed us to find a specific parts of the algorithm and migrate them to CUDA. Trying to launch a project directly on CUDA is a very risky task, since CUDA uses data that was defined and initialized on the CPU/host.

Despite what was said, this thesis can become a first step into making a more complex piece of software that can be used in CUDA. Knowing what works and what does not can be a significant step forward for future developers, choosing this task.

9 Summary

In this thesis we attempted to understand the basics of CUDA and attempt to optimize basic search procedures on two-dimensional arrays and on a NoSql library WhiteDB.

At the beginning we emphasized on how CUDA importance grows in IT industry. Showed main reasons why parallel operations performed on the GPU can be interesting for developers and scientists.

Described the difference between CPU and GPU architectures and demonstrated how parallel functions can be done in CUDA. Also showing the main parts of basic CUDA programming.

Tried to demonstrate how CUDA and databases can have a relationship. First focused on in-memory databases because of their architecture and usefulness. Reminded about two main approaches when it comes to row based and column based data and discussed some achievements in CUDA-based databases.

We demonstrated CUDA memory management and compiled experiments scenarios. With the acquired knowledge we attempted to use CUDA with a NoSQL library named WhiteDB.

All experiments and integration results are documented and analysed.

10 Pointer to code files

All the used code for this thesis is uploaded to github:

https://github.com/devrais/Andrei-Orehhov-Master-s-thesis-2017

11 References

[1] Nickolls, John, et al. "Scalable parallel programming with CUDA." *Queue* 6.2 (2008): 40-53.

[2] Garland, Michael, et al. "Parallel computing experiences with CUDA." *IEEE micro* 28.4 (2008).

[3] Bakkum, Peter, and Kevin Skadron. "Accelerating SQL database operations on a GPU with CUDA." *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*. ACM, 2010.

[4] Mivule, Kato, et al. "A review of cuda, mapreduce, and pthreads parallel computing models." *arXiv preprint arXiv:1410.4453* (2014).

[5] Gribble, Christiaan. "Introducing multithreaded programming: POSIX threads and NVIDIA's Cuda." *American Society for Engineering Education*. American Society for Engineering Education, 2009.

[6] “What is GPU-Accelerating computing? (2017)” [WWW] http://www.nvidia.com/object/what-is-gpu-computing.html

[7] “An Easy Introduction To CUDA and C++ (2017)” [WWW] https://devblogs.nvidia.com/parallelforall/easy-introduction-cuda-c-and-c/

[8] “CUDA C programming guide (2017)” [WWW] http://docs.nvidia.com/cuda/cuda-c-programming-guide/

[9] “An Even Easier Introduction to CUDA (2017)” [WWW] https://devblogs.nvidia.com/parallelforall/even-easier-introduction-cuda/

[10] “Memory Architecture (2017)” [WWW] https://cvw.cac.cornell.edu/gpu/memory_arch?AspxAutoDetectCookieSupport=1

[11] “Unified Memory in CUDA 6 (2017)” [WWW] https://devblogs.nvidia.com/parallelforall/unified-memory-in-cuda-6/

[12] Landaverde, Raphael, et al. "An investigation of unified memory access performance in cuda." High Performance Extreme Computing Conference (HPEC), 2014 IEEE. IEEE, 2014.

[13I lgner, R. G., and D. B. Davidson. "A comparison of the FDTD algorithm implemented on an integrated GPU versus a GPU configured as a co-processor." Electromagnetics in Advanced Applications (ICEAA), 2012 International Conference on. IEEE, 2012.

[14] “CUDA – Tutorial 4 – Atomic Operations (2017)” [WWW] http://supercomputingblog.com/cuda/cuda-tutorial-4-atomic-operations/

[15] Lake, Peter, and Paul Crowther. "In-memory databases." Concise Guide to Databases. Springer London, 2013. 183-197.

[16] Larson, Per-Åke, et al. "High-performance concurrency control mechanisms for main-memory databases." Proceedings of the VLDB Endowment 5.4 (2011): 298-309.

[17] Meixner, Matthias. "Main Memory Databases." (2005): 341-344.

[18] Kulkarni, Jyoti B., A. A. Sawant, and Vandana S. Inamdar. "Database processing by Linear Regression on GPU using CUDA." Signal Processing, Communication, Computing and Networking Technologies (ICSCCN), 2011 International Conference on. IEEE, 2011.

[19] Kanade, Anuradha S., and Arpita Gopal. "Choosing right database system: Row or column-store." Information Communication and Embedded Systems (ICICES), 2013 International Conference on. IEEE, 2013.

[20] “How Do Column Stores Work? (2017) ” [WWW] http://kejser.org/how-do-column-stores-work

[21] Pietron, Marcin, Pawel Russek, and Kazimierz Wiatr. "Accelerating select where and select join queries on a GPU." Computer Science 14.2 (2013): 243.

[22] Sahoo, Abhaya Kumar, Sundar Sourav Sarangi, and Rachita Misra. "A comparison study among GPU and map reduce approach for searching operation on index file in database query processing." Man and Machine Interfacing (MAMI), 2015 International Conference on. IEEE, 2015.

[23] Dean, Jeffrey, and Sanjay Ghemawat. "MapReduce: simplified data processing on large clusters." Communications of the ACM 51.1 (2008): 107-113.

[24] “PG-Storm Limit Breaker of PostgreSQL powered by GPU (2017)” [WWW] http://strom.kaigai.gr.jp/manual.html

[25] “High Performance GPU Data for Big Data SQL (2017)” [WWW] https://blazingdb.com/index.html

[26] “Meet the Future of Analytics (2017)” [WWW] https://www.mapd.com/products/

[27] “NVIDIA CUDA Compiler Driver NVCC (2017)” [WWW] http://docs.nvidia.com/cuda/cuda-compiler-driver-nvcc/#axzz4gwgrmYlU

[28] “MapD Core Database (2017)” [WWW] https://www.mapd.com/products/core/

[29] ”NVIDIA CUDA Library (2017)” [WWW] http://horacio9573.no-ip.org/cuda/group__CUDART__DEVICE_gb76422145b5425829597ebd1003303fe.html

[30] “What is a warp in CUDA? (2017)” [WWW] http://cuda-programming.blogspot.com.ee/2013/01/what-is-warp-in-cuda.html

[31] “WhiteDB (2017)” [WWW] http://whitedb.org/