CUDA DBMS

A study on the feasibility of GPU databases

2009

Software

Aalborg University

Morten Christiansen

Christian Eske Hansen

The Department of Computer Science

Aalborg University

Software Engineering, 10th semester

Title:
CUDA DBMS

Project period:
February 2nd 2009 -
June 10th 2009

Theme:
GPGPU Programming

Group:
d626a

Authors:
Morten Christiansen
Christian E. Hansen

Supervisor:
Bent Thomsen

Circulation: 4

Pages: 71

Appendices: A - B

Abstract

In this report we document the development of a GPU-based DBMS. It stores data on the GPU and uses the high computation power of the GPU to perform efficient parallel operations on it. The DBMS is written in C# for the CPU code and CUDA for the GPU code, and is accessed through a custom LINQ query provider.

The main building block of the DBMS is collections of data-parallel arrays which represent tables and table columns. To facilitate the standard database operations we have implemented CUDA kernels which perform set operations on the columns as well as various math operations. We have also implemented *cache sensitive search trees*, an index type which encodes a tree structure into an array for efficient data retrieval on the GPU.

Tests show that inserting values into the DBMS is in the order of a 100 times faster than SQL Server, and math operations were about 4 times faster. However, appending values to existing tables becomes increasingly slow with more values, and selecting an entire table also lags behind the SQL Server. Unfortunately, we were unable to perform filtered selections but performing tests directly on the index indicated it has a good performance.

While the implemented DBMS is only a prototype, there is a lot of promising work to continue with. This includes expanded support for types and operations such as strings and joins, as well as optimizations such as query restructuring techniques and more efficient memory management.

Preface

This report is written in the spring of 2009 by two software engineering students at the department of computer science at Aalborg University. In order to understand the report, a general understanding of programming and C# is required.
Source code presented in the report may differ from the original source code as some details may be left out to improve readability and/or to emphasize certain points. The source code as well as a digital copy of the report can be found on the enclosed CD. It also includes the necessary drivers and runtime components to execute the DBMS, given the presence of a supported GPU. A summary of the report can be found in Appendix A and a brief overview of the source code can be found in Appendix B.

Project group d626a

Morten Christiansen   Christian E. Hansen

Contents

Chapter 1

Introduction

While the increase in computation power of CPUs and GPUs has largely followed Moore's law, stating that the number of transistors in a CPU double every 18th month, it has become apparent that the processing power of CPUs have been limited by scaling issues for a single core, leading to the current shift toward multicore CPUs. The GPUs in contrast are massively parallel and have thus been free to scale performance close to linear to the amount of transistors [1].

To give an example of the difference in scaling, consider the Core i7 3.2GHz CPU and the Radeon HD 4870 X2 GPU. Both devices are, currently, among some of the most powerful of their respective processor types. While the CPU can perform a little over 51 GFLOPS it is not in the same league as the GPU which can perform 2,400 GFLOPS [2, 3]. The computation power of the GPU is a theoretical maximum and the practical value depends highly on the implementation and the type of computation but it is still a remarkable difference in scale.

In the last couple of years, *General Purpose GPU* (GPGPU) programming has become possible with new programmable GPU architectures, making the computation power accessible to programmers in general. This field has mostly been embraced for scientific computing but is slowly seeping out to more general programming areas as the tools and technology mature. The future of GPGPU programming is hard to tell, though, as there are several competing technologies and rapidly evolving hardware platforms. As CPUs and GPUs appear to be converging towards the same capabilities, a direction which is necessitated by the scaling issues with CPUs, we might be facing a future where many of todays assumptions and distinctions are no longer correct [4, 5].

When considering the difference in computation power, one has to keep in mind that CPUs have spent their transistors on other types of hardware than ALUs such as large banks of cache memory which makes them very suitable for general programming. The GPUs, however, are focused on similar instructions issued across a large set of data using *Single Instruction Multiple Data* (SIMD) operations, which makes it more difficult to take advantage of the full processing power.

One area that might benefit from this emerging technology is database operation as the analysis and operation of large sets of data is common in many fields such as scientific computing and business analysis. Previous research into databases leveraging SIMD instructions indicate that certain kinds of operations are highly suited for parallel instructions. For example, in [6] SIMD instructions on four values in some cases yielded a super-linear increase in performance.

1.1 CUDA Overview

This report discusses a technology which is not familiar to many programmers, so in this section we give an overview of the terms and concepts from CUDA programming which are used in the report. For a more detailed description of GPU architecture and CUDA we refer the reader to [7, 1].

CUDA is a parallel computing architecture which allows programmers to write general purpose programs that run on the GPU, thus taking advantage of the high computation power of modern GPUs. GPUs are built using a massively parallel architecture using wide SIMD instructions.

To program for the GPU architecture, the programmer has to distinguish between code running on the GPU and code running on the CPU. These two areas of execution are also referred to as the *device* and the *host*, respectively. On the device, small programs, or *kernels*, are executed in parallel. These kernels are written in C for CUDA, which is an extension to the C language, and the form of a kernel is to write the execution of a single light–weight thread. In the host code you specify how many threads should run a given kernel. In CUDA, threads are grouped in *blocks* arranged in *grids*. This gives a logical structure to thinking about the threads but it also reflects the nature of the hardware where a block of threads share a certain amount of resources. Figure 1.1 illustrates this structure and how it maps to the GPU memory hierarchy. Similar to the memory hierarchy on the host, the more local memory banks are faster though considerably smaller.

Listing 1.1 shows an example of a CUDA kernel along with the code invoking the kernel from the host. In the kernel, the variable `i` is set as a unique value representing the executing thread, based on the id of the executing thread and block. This means that thread `i` accesses the `i`th value of the arrays `x` and `y` to calculate a new value for the same index in `y`. When calling the kernel, the parameters in the angle brackets determine the amount and size of thread–blocks to start.

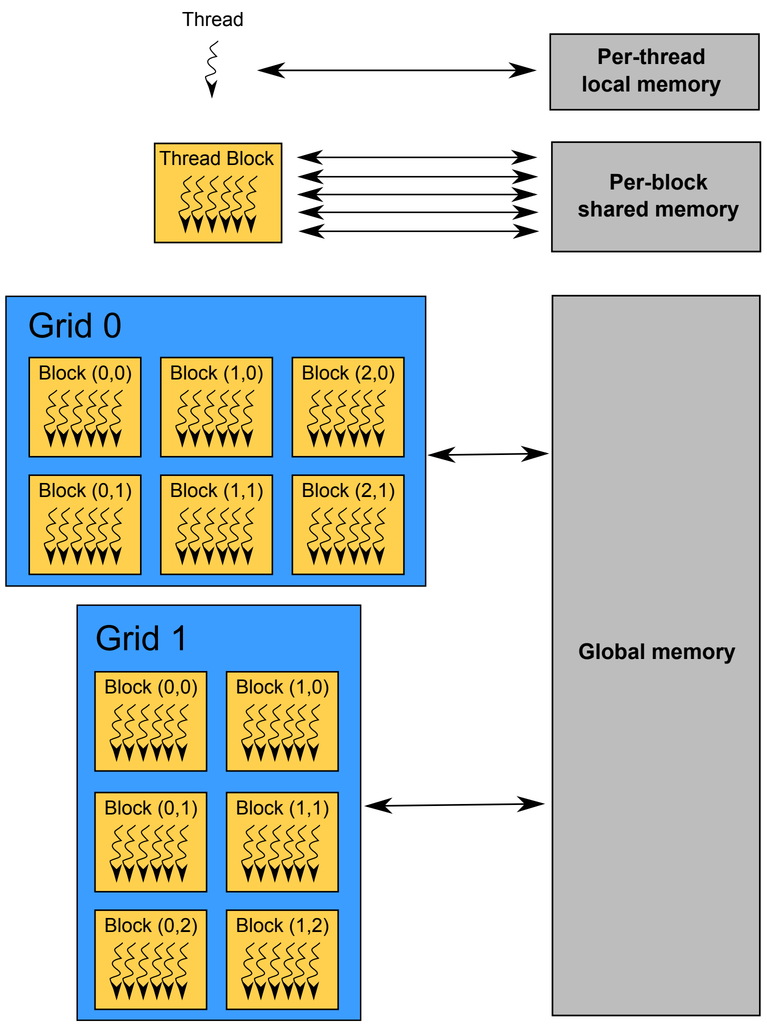

Figure 1.1: The GPU thread hierarchy and how threads access the memory hierarchy.

```
//Kernel definition
__global__ void
saxpy_parallel(int n, float alpha, float *x, float *y)
{
   int i = blockIdx.x * blockDim.X + threadIdx.x;
   if (i < n) y[i] = alpha * x[i] + y[i];
}

//Invoke parallel SAXPY kernel (256 threads per block)
int nblocks = (n + 255) / 256;
saxpy_parallel<<<nblocks, 256>>>(n, 2.0, x, y);
```

Listing 1.1: Scalar Alpha X Plus Y algorithm example [8]

A block can contain up to 512 threads but it is executed in groups of threads called *warps* which contain 32 threads. These threads are executed synchronously on a multiprocessor which takes one instruction at a time and executes it for each thread on individual pieces of data. If the program flow branches, however, the GPU can only execute the threads on one branch at a time.

As the device works on large amounts of data, it becomes essential for performance to retrieve enough data from memory to fill the multiprocessors. This is done by using a couple of techniques for accessing data in parallel. To efficiently fetch data from *global memory*, which is the equivalent to main memory on the host, you can use *coalesced accesses*. If you obey certain rules for memory alignment and data sizes, a coalesced access allows you to retrieve 32 values in a single memory fetch. For shared memory, which is local to a given thread block and which can be considered as a small manually maintained cache, a similar option exists. The shared memory is distributed across 32 physical banks which can each service a single request at a time. Thus, to achieve optimal memory throughput you must fetch 32 values which are adjacent in memory and aligned to the banks.

GPU memory is high bandwidth but also high latency, so in order to mask this latency, the GPU switches between the execution of different blocks instead of blocking while waiting for memory fetches to complete. While the thread inside a given block can be considered to be executed in–order, this has the effect that different blocks are executed entirely independently from each other and in an arbitrary order. It is possible to communicate between blocks using global memory but this has all the normal dangers of concurrent programming with shared state. To handle this, the GPU provides a mechanism for synchronizing threads using a thread barrier.

Chapter 2

Problem Statement

In [7] we explored the possibilities for creating a GPGPU library which could provide an interface to a number of CUDA operations from .NET languages. Having gained an overview of the obstacles involved in building a GPGPU API, we take a step back and consider the implications of our experiences and how to leverage them for a better API. In this chapter we describe the most prominent issues we face and we present possible solutions to them, in high–level terms.

2.1 Main Problems

One of the problems we had to deal with in our previous work was when performing computations on the GPU, where data has to be transferred back and forth across the PCI–Express bus. This is a bottleneck which dominates the performance benefits of simple operations for all but the largest input sizes [7]. This problem made it necessary to either evaluate each request to the library to find the best place to perform the computations or simply rely on the users of the library to benchmark the performance of their code. Concrete examples of the overhead associated with transferring the data are in implementations of join, radix sort and hash search operations of which the cost of the total execution time is 0–13%, 10–30% and 40–60% respectively [9, 10].

One of the purposes of the library we worked on was to create a mapping of programming concepts between ordinary .NET programming and GPGPU programming. The way we approached the problem was to create a wide range of computation providers for many different problems and cases. This design makes it hard to use the library in a more general and flexible way and it limits the use to the set of functions we, as library designers, could think of to implement. The cause for this strategy was that we did not find a good way to represent more complex data than the simple types. This limited us to general math operations on simple data.

The nature of current GPU hardware poses one problem which we cannot avoid, but hopefully mitigate, and that is the data–parallel programming model required by the hardware. To achieve anything near the potential performance of the GPU, the computations we perform must adhere to these hardware limitations.

2.2 Proposed Solution

One of the results of our previous work was the idea of having data living on the GPU for most of its life cycle. The major motivation for this approach is the benefit in improved locality between the processing resources and the data on which they operate. If it is feasible to expect that most data can be stored effectively on the GPU we will have solved one of the major bottlenecks in applying the potential of the GPUs. To determine whether this is the case we must take into consideration the extra cost of storing and operating with irregular data on the GPU.

The idea of having data reside long–term on the GPU also sparked ideas of extending this to include a database–like data representation and operation interface. An issue which a DBMS would address is the representation of complex data and relationships as well as computations on them. The relational model of representing data and relationships between them is used in most DBMSs and is well documented and tested. Using this model to represent data on the GPU provides a model that is not foreign to developers and which allows them to describe more complex interactions than our previous approach. A benefit of this representation of data is also that we can perform more complex operations over the data. Instead of forcing the developer to think in terms of operations on arrays of simple types, he can think on a higher level of abstraction, for example to define new entities as a result of operations on other entities. This ability is especially useful if combined with standard database operations for creating groups of entities and relations with specific properties.

If we store data as entities and relations it is an obvious step to extend the functionality to standard database operations such as `SELECT`, `JOIN`, `DELETE`, etc. In combination with math operations these make it possible to perform very complex operations on data. This flexibility is essential if the DBMS is to be useful in a general development context. One issue these capabilities can introduce is operations on non–linearly allocated data which reduces the memory bandwidth on the GPU. We must determine to what extent this is a problem and provide an overview of which types of operations it is worth performing on the GPU. While database operations are useful metaphors for performing complex operations on data, previous work also suggests that there is a lot of performance potential in calculating certain database operations on the GPU such as joins, relational queries and range queries [9, 11].

It is not enough to have complex data storage and computation capabilities in the DBMS if taking advantage of them is not easy and well integrated with the host language. To solve this problem we look at the possibility of using a custom LINQ provider as our main interface to the DBMS. This provides type–safety and Intellisense help, thereby allowing developers to maintain a high level of productivity. A LINQ provider already exists for interacting with relational databases in LINQ–to–SQL and this model provides for a useful source of inspiration as ours would solve essentially the same problem only in a different representation.

An important factor with regards to the core DBMS design is the GPGPU platform it is built on. In [7], we argued that the different languages are built with similar design philosophies, as they are partly dictated by the current GPU hardware architectures. Based on this observation and the fact that a defacto standard has yet to emerge in the field of GPGPU programming, we focus on the general design and assume that CUDA can relatively easily be exchanged for a different language should this prove necessary in the future. One of the main results of this project is to find a solution which can work with the GPU architecture and which is easy and intuitive to use – these aspects would be the same regardless of which technology we would use and, therefore, we use CUDA which we have previous experience with.

The above solutions proposed to solve the problem described in Section 2.1, leads to the question we try to answer in this project:

Is it possible to develop a DBMS residing in GPU memory using CUDA to facilitate a flexible data representation and computation model which can be accessed through a LINQ query interface, and if so, how does its performance characteristics compare to existing general purpose DBMSs for different types of tasks?

2.3 Exploring a Use Case

To illustrate how the DBMS could provide a simple and easy way to solve different computation problems, we provide an example use case.

Consider a museum which tracks the position of its guests at a certain interval using RFID tags, each with a unique identifier. At the end of each day, many thousands of positions have been entered into their database. What the museum needs is an efficient way to identify patterns in this data. For example, they might want to colorize a map of the museum by the amount of time visitors spent in each area, so that areas which keep visitors for long are marked with red colors while areas with less attraction are marked with cooler nuances. This would provide the museum with a popularity map of their expositions which they could use to determine when to retire expositions in favor of something more interesting.

Lets consider the computational task the above scenario introduces. A simple way to map the problem would be to organize the museum map into a grid of square cells and then count the number of positions registered in that area and divide it with the number of unique identifiers making up the positions. This will give a measure of the length of time people on average spend in a given square. This is, of cause, not useful if we consider all the data at once, so we need to separate the popularity measures by their time of occurrence such that you, for instance, can get a color map for each day or week.

Listing 2.1 shows how a LINQ query could be composed to calculate a popularity map of the museum. It is completely transparent, from a syntax point–of–view, that you are in fact interacting with a database and not just an ordinary object collection. There is no guarantee, though, that you would not have to know something about the internal implementation to get the most performance out of the DBMS.

```
public class Measurement
{
    public float X { get; set; }
    public float Y { get; set; }
    public DateTime Time { get; set; }
    public int UID { get; set; }
}

public IEnumerable<float> GetPopularityValues
    (
        Table<Measurement> measurements,
        DateTime fromTime,
        DateTime toTime,
        float cellEdgeLength,
        gridWidth
    )
{
    return from m in measurements
           where m.Time > fromTime && m.Time < toTime
           let cellId = (int)(m.X / cellEdgeLength + gridWidth * m.Y /
               cellEdgeLength)
           group m by cellId into cells
           let uniques = (from msr in cells
                          group msr by msr.UID into idGroups
                          select idGroups.Count()).FirstOrDefault()
           select cells.Count() / uniques;
}
```

Listing 2.1: A LINQ statement querying an instance of our DBMS for positional data.

Using LINQ as the way to construct algorithms allows the developer to describe his problem in high–level declarative code without worrying about such issues as how to iterate over collections of data, access indices, and which parts of the algorithm can be executed in parallel. This simplicity also reduces the amount of work required to change or make variations of an algorithm. In the example above, the problem we addressed was just one out of many potential issues so it is a desirable quality to be able to easily create a wide range of queries and be able to get an output as fast as possible. Other examples which could be relevant for the museum case would be to implement the algorithm for calculating the trajectories of moving objects in tracking networks described in [12] or the algorithm for optimized WLAN positioning described in [13]. Both of these problems revolve around analyzing large amounts of numeric data, a task for which GPUs are highly suited.

We hope that combining the clear and high–level abstractions of the LINQ interface with the computational power of the GPU allows us to achieve this.

2.4 Related Work

The subject of GPGPU programming is a new subject and the field is only just beginning to establish its identity. As such, the amount of research and work put into derivative technologies, such as GPU–resident DBMSs, is limited. We have thus been unable to locate any working solutions on this specific topic though an ongoing project addresses the subject in a similar but slightly different way. The *GPU Query Co–Processing* (GPUQP) [14] project is described as 'a relational query engine that employs both CPUs and GPUs for in–memory query co–processing'. Only a brief overview has been published on this project but its purpose is described as an add–on to ordinary DBMSs for sending selected database operations and the related data to the GPU where it is processed and then transferred back to the database. The people behind the project are working on implementing several database operations efficiently on the GPU using a set of primitives such as map, scatter, gather and sorting operations. They have currently published their work on implementing join operations using these primitives [9]. Their experiences from this work suggest that the primitives are a good basis for database operations and that they not only apply for the join operation.

Figure 2.1 shows the topology of the major research areas as they relate to our work, which is represented by the asterisk in the middle. Our work attempts to bridge the two fields of databases and GPGPU programming and there are a number of technologies in the two respective fields which are particularly related to our work. The figure also includes the field of general programming as this is the context in which the DBMS is intended to be used.

Figure 2.1 also serves to point out the major aspects of the DBMS we need to work on. The overlap between the database and the GPGPU fields covers the representation of the database on the GPU and the interaction between the high level, non–GPU code and the

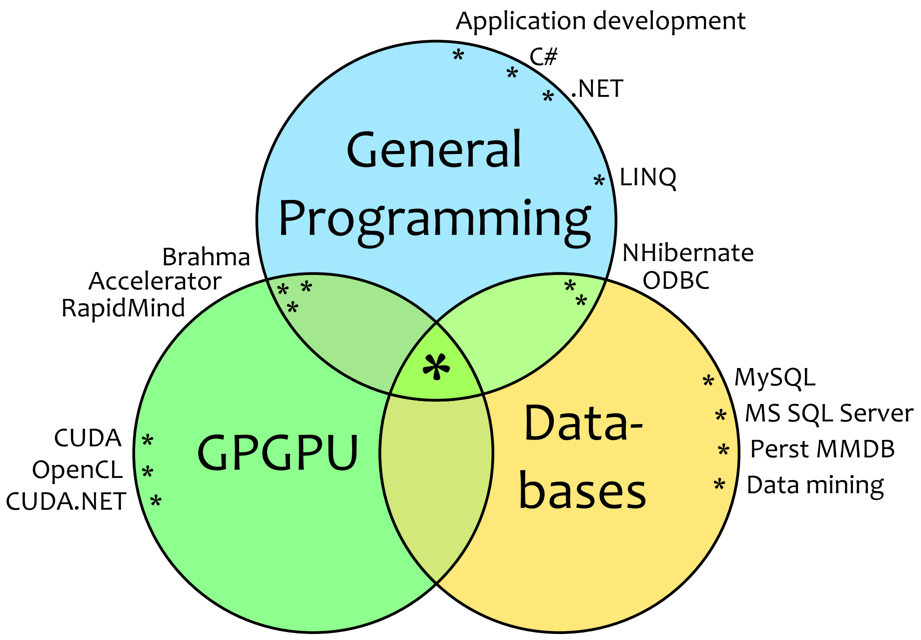

Figure 2.1: How the different fields of study overlap.

low–level CUDA code. This part is internal to the system as a whole and users of the DBMS do not have to concern themselves with its details. The overlap between databases and general programming covers the creation and querying of the database and as such is the public interface to the database. The overlap between general programming and GPGPU represents the limitations imposed by the GPU hardware on how computations must be performed to achieve high efficiency. This is an important subject as it can make the DBMS more difficult to use if not properly communicated through the API and our goal is to minimize the level of awareness of the hardware platform required to use it.

From the field of databases there is a lot of work to draw on but a specific subject which is particularly aligned with the design principles of a GPU DBMS would be *Main Memory Databases* (MMDBs). These are also designed to work on data in a low latency, volatile storage environment and can provide good insight into how to store and access data. It can also provide us with an overview of problems and difficulties in taking this approach. In chapter 3, we discuss this topic in greater depth.

From the field of GPGPU programming, several libraries exist which serve to make GPU resources available for general programming by creating abstractions on top of the GPU programming model and integrating with managed languages. They generally focus on a single abstraction to interface with the underlying platform. Below is an overview of some of these libraries and their principle abstractions as well as our impression of their usefulness for our work:

- Accelerator is a research project from Microsoft which provides a number of data–parallel array structures in .NET [15]. The main concept is that you turn a normal array into a parallel version, perform a number of operations on it, and turn it back to a normal array when you need the data. The library analyzes the computation graph of operations on the arrays, and performs a number of optimizations. Depending on our implementation these optimizations could apply to our work as well and we keep them in mind during the design of the DBMS.
- Brahma is an open source library which works by translating LINQ queries on instances of data–parallel arrays into shader code which can be executed on the GPU [16]. It tightly integrates the operations on the CPU with those to be performed on the GPU by providing a custom query provider. Not much documentation exists for this project but by looking at the code we might be able to get inspiration on how the LINQ programming model can be applied for different types of operations.
- Like the Accelerator library, Rapidmind provides an array–abstraction as the way to perform operations on data in C++. The strength of the Rapidmind platform is that it has load balancing and distribution of work across heterogenous hardware resources such as CPUs, GPUs and the Cell processors found in the PlayStation 3 [17]. Rapidmind is a closed, commercial platform which makes a deeper analysis difficult, but its most interesting aspect would be the flexibility to use different hardware. While this would be an interesting subject for a hybrid DBMS, the focus of this project is a purely GPU–located DBMS and other hardware resources are not taken into consideration.

The problem with these solutions is that they provide a simple model for interacting with data. The data–parallel array model they use allows for many operations on simple data types, but for complex operations you have to provide the necessary abstractions yourself.

The Brahma library provides a more flexible computation model as it uses the LINQ query API to query the arrays but it requires creating unfamiliar scaffolding code to setup and use the computations you define. Currently, there exists no documentation for Brahma, which makes using it difficult, though a couple of example applications exist.

All of the libraries lack any sense of relationships between entities and still revolve around performing a given operation across all members of an array of simple types.

Chapter 3

DBMS Design

This chapter describes the architecture and design of the DBMS, as well as the overall functional requirements for it. The design is driven by the problems and proposed solutions described in Chapter 2, drawing inspiration from existing DBMS designs.

3.1 Design Goals

The most obvious goal for the DBMS design is to solve the problems described in Section 2.1 which are directly concerned with performing operations on data in an intuitive and efficient way. As we are developing a DBMS it becomes interesting how it performs compared to existing DBMSs in a more general context.

If we design for generality to compete with existing DBMSs we have to take a more holistic view of the environment in which the DBMS runs including the ACID properties and acceptable performance in all areas. While a desirable goal, these requirements do not fit well with the GPU architecture as it is specialized hardware. Thus, the main goal of our design is to develop a specialized DBMS which is good at certain use patterns. We look at the use patterns which lie in the intersection of heavy computations and traditional database tasks such as data–mining. In Section 3.3, we go into further detail of how the GPU architecture influences the preferred use patterns for the DBMS.

We maintain the secondary goal of determining the applicability of the DBMS as a general DBMS, although more from a theoretical viewpoint, and try to provide an overview of the long–term prospects of this goal.

Focusing on a design philosophy of a more specialized DBMS, we design it to achieve a high level of performance for the most suitable tasks. This means that the choice of indices and how data is represented might result in very poor performance for certain operations but instead of addressing this directly we provide an analysis of potential ways this could be solved.

3.2 Implications of the GPU Architecture

This section describes how the GPU hardware affects the DBMS design. It is important to carefully consider how you interact with the GPU as current GPU architectures are highly optimized for certain types of operations. Because of this difference in architectures between CPUs and GPUs [18], we cannot apply an existing DBMS design and are forced to reconsider conventional wisdom in the field. This is primarily due to the way data must be fetched from GPU memory to achieve a high throughput.

Below is an overview of how the GPU architecture impacts how we design the DBMS.

Recomputations Over Fetching Values As the memory of GPUs have a high latency[1] it can be considerably faster to recalculate a value than fetching it from global memory.

Maximize Memory Throughput To achieve a high memory bandwidth to the GPU arithmetic units from shared and, especially, global memory, we must take advantage of coalesced access, i.e., the ability to retrieve values for an entire half–warp in a single instruction. In this aspect, the data access pattern of a GPU DBMS lies between those of the disk–based and the memory–based DBMSs. This means that we can perform random accesses to GPU memory but in blocks of data, as on a hard disk, only smaller.

Minimize Kernel Calls Each time you execute a CUDA kernel, you pay a little overhead for initializing the code and starting the GPU pipeline. To avoid this cost we must keep the amount of kernel invocations from the host to a minimum. Instead we can group together several queries and encode a set of operations in a format which a single kernel can process. An added benefit of performing fewer and larger operations is that the use of shared memory can be optimized. As shared memory access times are about a factor 100 faster than those of global memory this can result in a significant performance difference[1].

Limited Memory Compared to *Disk Resident Databases* (DRDBs) and even MMDBs, the storage space available for a database on the average current GPU is limited. Currently, the most memory available on a standard commercial graphics card is 4GB but the average for a CUDA–enabled card is perhaps closer to 512MB if not lower. While this amount is constantly increasing it does make for a problem if the data–set you work on is larger. This makes it necessary to consider some means of swapping data between the GPU and main memory.

Keeping Non–Data on the Host The GPU is geared towards data–parallel operations and therefore it is wasteful to perform serial operations on the GPU. Managing indices, queries and other parts of the DBMS can require a lot of serial operations which are best suited for the CPU to process so we must keep as much as possible of these types of operations on the host. However, once a kernel has been invoked, it might be faster to recompute some value rather than interrupting the work flow to get it from the host. It can, though, be faster for some tasks to rethink their implementation and create parallel versions. Indices, for example, could benefit from being accessed in parallel.

Floating Point Precision A potential problem with the current generation of GPUs is their lack of support for floating point operations as specified by the IEEE 754 standard [19]. This standard is a widely accepted standard and is, for example, used in Intel CPUs [20]. For most floating point operations, CUDA deviates slightly from the standard [1], but this lack of precision can be ignored in many cases, depending on the application area.

Another related issue is the lack of support for double precision numbers in GPUs with a compute capability version lower than 1.3, which is a hardware specification describing how well a given GPU supports CUDA[1]. On these GPUs, double precision values are reduced to single precision.

As these issues are a result of the hardware implementation there is nothing we can do to correct them but as GPGPU programming gains momentum and as older GPUs get replaced, these problems will become less of an issue but they do impact the usefulness of our DBMS for certain applications.

3.3 DBMS Requirements

In this project we explore how well the GPU architecture is suited to host a database and how we can integrate it with .NET application development. As mentioned in Section 3.1, our design goals are to create a specialized DBMS which leverages the benefits of the GPU hardware to maximize performance for a set of use cases. We use the project to gain practical and detailed experience with this DBMS design, from which we then put the DBMS in a greater perspective – comparing it with other types of DBMS and discussing how it can be extended to embrace a more general use pattern.

The requirements put forth in this section reflect this core functionality which we believe the DBMS is most suitable for. The main goal this functionality must achieve is to provide a general representation of data and a flexible model for operating on it.

Based on the GPU hardware, the general use case for the DBMS has the following characteristics:

- Computation-intensive operations
- Limited size of data-set
- Large scale parallel operations
- Volatile storage

These characteristics point toward a preferred environment with few but computationally expensive transactions. This makes the DBMS less suitable for a multi-user environment with lots of small updates and for serving static content. The high computation power and volatile memory suggest a better use case would be just-in-time computations and analysis on large amounts of data. Such cases include data-mining where users desire responsive visualization of the data based on a large amount of parameters or a stream of data which must be analyzed in real-time.

The focus for the DBMS described above leads to the following concrete limitations of the scope of the DBMS, as implemented in this project:

Single-user DBMS To design the DBMS with a focus on few but large operations, we make it a single-user DBMS. The DBMS provides a local in-process database which works on the same thread as it is instantiated, providing a single connection to query against.

Lack of ACID properties Durability is a problem for all MMDBs as the data lives in volatile memory. While some DBMSs such as the Oracle TimesTen DBMS use logging and checkpoints to create backups in non-volatile storage [21], we keep our focus on use the use cases where data loss is less of a problem and performing computations on more static data is prevalent.

As only a single connection to a database instance is possible at a time, atomicity and particularly isolation become less relevant. While it is possible to introduce query optimizations which would make isolation and atomicity important, the inherent lack of threading issues makes it possible to avoid most of the issues around accessing inconsistent state. Atomicity is also useful for maintaining consistent data in the case of a database failure but due to the lack of focus on durability, the inconsistent data will be lost in any event.

Support for a subset of Database operations To explore the feasibility and performance of a GPU DBMS, we focus on the key database operations and do not try to implement the full complement of standard database operations. This focus is on the basic operations in relational calculus[22] as well as some common math operations.

3.4 DBMS Architecture

This section describes the major components of the DBMS and how they are tied together. To identify the main components of a DBMS we look at the architecture of the ALTIBASE MMDB[23]. In the article they enumerate the standard components in a DBMS which include index, recovery, transaction and memory managers as well as a query processor and a library of native stored procedures. As mentioned in Section 3.3, we do not focus on multi-user support and data durability and, hence, the transaction and recovery modules are not interesting for us. The remaining four components encapsulate the main functionality of our DBMS, and how they communicate is illustrated in Figure 3.1. The query provider module is the interface to the DBMS.

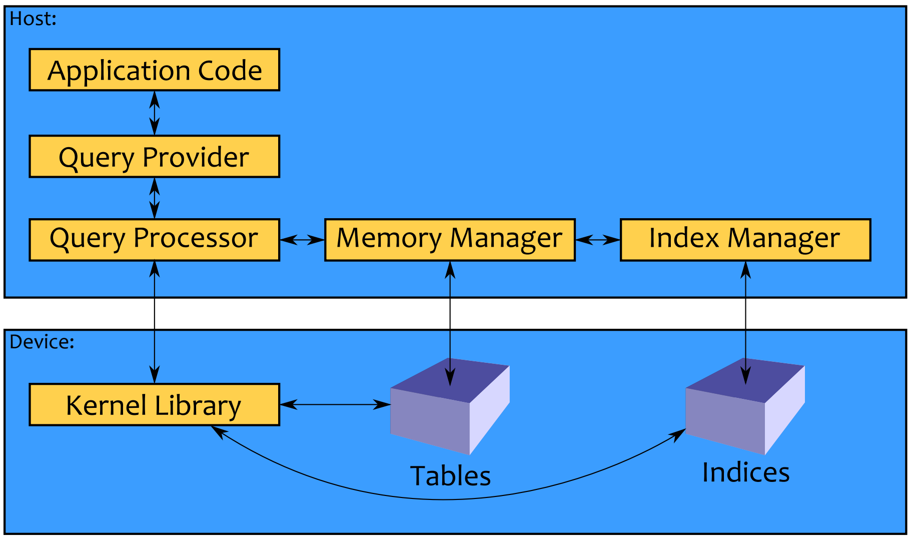

Figure 3.1: The high level architecture of the GPU DBMS.

Our equivalent for the library of native stored procedures is a library of CUDA kernels which can perform all the different database operations we need.

The modules shown in Figure 3.1 are described in detail in the following subsections, each with a table describing their abstract interface. This interface enumerates their main responsibilities as well as the forms of data they consume.

3.4.1 Interface

Interface function	Return value
CreateQuery(LINQ Query)	void
OutputResult(data stream)	objects

The public interface is the interaction point from which users of the DBMS can set up and query a database instance. There are many standards for database connections such as ODBC which would allow access to the database from different programming languages and environments but for the scope of this project we focus on a single interface by implementing a custom LINQ provider for a strong and type-safe integration with .NET languages.

Using a LINQ provider as the interface to the DBMS makes it possible for us to perform lazy evaluations on queries. When a query is first made, the result is just a representation of the query, such as SQL. It is only when you iterate over the resulting `IQueryable` object that the query is executed. This makes it possible to compose complex queries by making additional queries on the original `IQueryable` object. Depending on how the query is performed, there are potential optimizations which we can use as knowledge of the situation increases.

3.4.2 Query Processor

Interface function	Return value
ExecuteQuery(query representation)	data stream

It is the job of the query processor to manage the execution and optimization of database queries and as such is closely connected to the implementation of the custom LINQ provider. Through this provider it can be used to transform a LINQ query into a representation which our DBMS can process on the GPU. As we are creating a DBMS it seems natural to represent queries as SQL since it can describe all the common database operations by design. As such, we use SQL as the basis for representing queries but keep an open mind for adding addition constructs if they make it easier to represent useful operations suitable for GPU hardware.

There are two approaches to executing queries on the GPU; sending a representation of the query to a kernel which orchestrates each of the sub-operations or managing each part of the query from the host as a series of kernel invocations.

Making a single kernel invocation for a given query would appear to be the optimal solution since this avoids the extra overhead associated with performing the invocations.

However, there are several issues which make this approach infeasible for many database operations. One of the main problems is that current GPUs do not support dynamic allocation of memory during a kernel execution. This means that operations such as joins where the output size is unknown must be performed in several passes or by pre-allocating the worst-case output size. For joins, the latter option increases beyond the GPU memory size for moderately large input sizes.

Several of the computation primitives used to implement database operations such as scatter and gather operations can also be optimized using multi-pass algorithms for caching [10].

3.4.3 Memory Manager

Interface function	**Return value**
CreateTable(schema, size)	void
DeleteTable(table)	void
StoreData(data, table)	void
RetrieveData(table)	data stream

The memory manager handles the allocation and consumption of GPU memory resources. It determines how data is structured as well as how meta-data, such as the database schema, is represented and maintained. In this section, we describe database relations from a C# view. This means that instead of using the terms relations and attributes, we use types and members instead.

The way entities are represented in the DBMS is as a schema and number of data arrays. The schema is a mapping from the name of a type and member(s) to a given array of values, where members correspond to attributes in relational databases. As an example, we use a type *Person* with the members *Name* and *Age*, there would be two arrays of data, one for each member. An instance of the type *Person* is thus comprised of the array elements with identical indices. Table 3.1 illustrates this mapping for *Person*.

Member name	**Data array**
`"Person.Name"`	`char[] pointer`
`"Person.Age"`	`int[] pointer`

Table 3.1: Schema for mapping strings to data for a simple type.

The *Person* type described above is a simple relation; to illustrate a more complex type consider that *Person* also has an *Address* member which, itself, is a type with the members *City* and *Street*. The most simple way to represent this relationship would be to add the members of *Address* as members of *Person* as illustrated in Table 3.2.

Member name	**Data array**
`"Person.Name"`	`char[] pointer`
`"Person.Age"`	`int[] pointer`
`"Person.Address.City"`	`char[] pointer`
`"Person.Address.Street"`	`char[] pointer`

Table 3.2: Schema for mapping strings to data with relationships between types.

This representation of relationships fits well with one-to-one relationships but less so with other relationships. For many-to-one relationships, for example if many people live on the same address, this will result in wasted space as data is represented redundantly. A better way to achieve this is to use a table describing the relationship in the same way ordinary relational databases handles this. Thus, to represent this relationship we need three groups of arrays, one group for each of the two types as well as a group with two arrays for the association. The latter group contains two arrays with the indices of related *Person* and *Address* entities.

Storing type members separately from its parent type impacts the database design in a number of ways. When you access a member of a child member, for example the *City* member from Table 3.2, the schema cannot map directly to a data array as the index of the *Person* entity does not match the index in the *Address* member arrays. Instead it must first look up the member in a translation table which translates the index if a relation is found as illustrated in Table 3.3.

Member name	**Index Transformation**	**Data instance**
`"Person.Name"`	`i` ⇒ `i`	`char[i]`
`"Person.Age"`	`i` ⇒ `i`	`int[i]`
`"Person.Address.City"`	`relation[i]` ⇒ `j`	`char[j]`
`"Person.Address.Street"`	`relation[i]` ⇒ `j`	`char[j]`

Table 3.3: Schema for mapping strings to data by transforming the array index to matching relation arrays using a translation table.

Another result of this storage scheme is that it becomes possible to sort each of the entities on a member independently as a means to achieve a better data locality and, thereby, be able to better take advantage of the GPUs parallel memory fetching techniques.

Alternative Data Storage Model

As described in Section 3.2, data must be stored linearly in memory to achieve the optimal memory throughput, but only for the size of the data being accessed by a given warp. It is possible to create a data structure which chains a number of linearly allocated blocks together, using the same principles as for linked lists.

If all the data in a given database column is stored sequentially in an array, deletions can fragment the data, which reduces memory throughput, and it is expensive to reclaim the memory. However, if the chained data structure is used, the data can be compacted by merging or deleting chain blocks. With the array structure it is necessary to scan the entire array, reallocate space enough for the entire set of data, and copy the data to the new array. An additional benefit with the chained structure is that it also becomes cheaper to maintain sorted arrays, due to the reduced cost of performing deletions and insertions.

The drawback of the chained structure is that it introduces a performance overhead as it has to follow references to find the proper blocks, an operation for which the GPU is not able to use coalesced memory access. In other words, the array structure is ideal if the size of the data set is static and the chained structure is better for more dynamic data sets.

Based on the use case described in Section 2.3, we focus on the simpler array structure, but keep in mind the possibilities for users to specify the storage format they prefer.

3.4.4 Index Manager

Interface function	**Return value**
CreateIndex(table, type)	void
UpdateIndex(table)	void
DeleteIndex(table)	void

The index manager handles the creation and maintenance of indices on the database tables.

Much work has been done in the area of database indices, both for disk- and memory resident databases. This includes arrays, B–trees, B^+–trees, hashes, T–trees, CSS–trees [24, 25] and CSB^+–trees [26]. For this project, the CSS–tree is of particular interest as it has previously been used to implement a parallel index on the GPU [14].

In this section, we describe a selection of possible data structures to be used for indexing. It is not a complete list of all alternatives, only including the ones likely to perform well. Binary trees are omitted, for example, as the small node size would cause low performance due to poor data locality.

Index Structures

A number of criteria are interesting when determining which data structure to use for indexing.

Time Complexity According to our requirements specified in Section 3.3, the index structure should primarily facilitate fast searching, but should still handle insertions and deletions reasonably well.

Space Utilization It is preferable to have as little space overhead as possible for two reasons. The GPU has a limited amount of available memory compared to MMDBs and DRDBs, so a smaller index means more room for data. Space overhead for pointers also leads to a performance decrease as less actual data can be retrieved in a single coalesced memory access.

GPU Mapping It is also important how well the index structure maps to the GPU hardware. This means utilizing optimization techniques available on the GPU.

Resizing a data structure to accommodate inserts and deletes is problematic on the GPU as memory cannot dynamically be allocated within a kernel. A way of dealing with this is to assign much more memory than needed and free surplus memory. However, as comparatively little memory is available on GPUs, a better way is to assign more memory blocks and distribute the data structure over these, using pointers as references. This works quite well on structures such as trees and hashes, but less so on structures that rely on arrays such as CSS–trees as this compromises the beneficial memory access pattern of arrays.

Arrays can be handled by maintaining a copy of the structure in main memory and copy this to the GPU when it is modified.

Array An array is a very space–efficient data structure as there is no space overhead for storing pointers. Deleting is quite expensive as the array must be rebuilt to prevent fragmentation, so arrays are most useful in situations where the DBMS is primarily used for querying existing data. Another problem is with searching the array. If the array is too large to fit into one coalesced memory read and binary search is used, memory bandwidth utilization is very low as only one value per coalesced read is used. Linear searching of the array has better memory bandwidth utilization, but has poor time complexity.

Chained Hashing Many types of hashing exist beside *Chained Hashing* (CH), but it is generally more efficient for indexing than, e.g., linear hashing [24]. One way of implementing CH on a GPU is to adjust the bucket size so that it fits well with coalesced access. While this requires a potentially large number of buckets, and thus some space overhead, it provides a very fast way of searching. However, an uneven distribution of data across buckets would decrease performance, and there is no efficient way of doing range queries using a hash index. This requires a secondary index to be maintained.

T–Tree T–trees have been proposed as an improved index structure for MMDBs [24]. A T–tree is a balanced, binary tree where each node contains a number of sorted elements and two pointers to the node's left and right child. Searching is done by starting in the root node and checking if the value is within the node. If the value is less than the left–most (smallest) element in the node, go to the left child, and vice versa for the right child, and repeat until the value is found. The primary problem with T–trees is that it is a structure which maps quite poorly to GPU hardware as it is not a "linear" structure such as arrays. Also, if the size of each node corresponds to a coalesced read, memory utilization is quite poor as each node not containing the requested value only has two interesting nodes; namely the left– and right–most nodes. Node utilization is naturally higher in case of ranged queries.

B^+–Tree B^+–trees are of more interest to us than B–trees. There are two reasons for this. Firstly, B^+–trees perform better than B–trees when performing range queries due to the pointers from leaf node to leaf node in B^+–trees. As the GPU is massively parallel, searching for single elements in the database is likely not the common use case. Secondly, a B–tree has more space overhead than B^+–trees as leaf nodes in B–trees contain null pointers. As a high percentage of nodes in a B–tree are leaf nodes, the overhead becomes increasingly substantial. Regarding mapping a B^+–tree to GPU memory, the size of each node would ideally correspond to a coalesced read.

CSS–Tree A CSS–tree is a modified, cache sensitive B^+–tree, which is a two–part array consisting of a directory structure and a sorted array. The directory is essentially a balanced search tree stored as an array. One particular advantage of this is that the tree does not have to store pointers to nodes. Traversal can be done by performing arithmetic operations on array offsets, which makes CSS–trees very space efficient. The specifics on this is explained later in this section.

Figure 3.2 shows an example of a *5–ary* CSS–tree with four keys per node. The internal nodes (i.e., nodes 0–15) contain keys for navigating to the correct leaf nodes. The leaf nodes (i.e., nodes 16–80) contain the values of the original, sorted array `a_data`.

Figure 3.3 shows the array representation of the CSS–tree from Figure 3.2 and the array `a_data` from which the tree is built. Each element in array `a_data` represents four index values in order to make the mapping between `a_data` and the CSS–tree nodes easier to visualize, i.e., the 0th element in `a_data` contains index values 0–3. Notice that the elements 50–64 in `a_data` correspond to nodes 16–30 in the tree. This is necessary as CSS–trees store

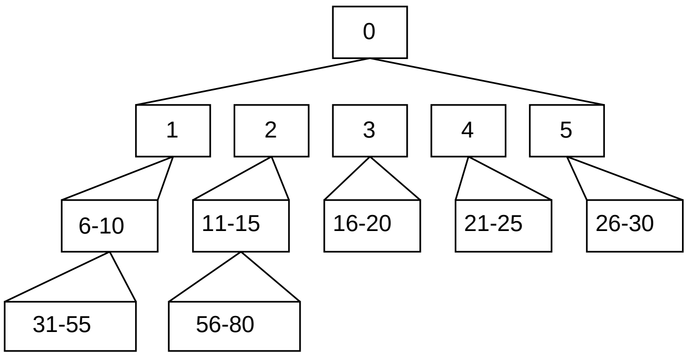

Figure 3.2: Example of a 5–ary CSS–tree. Each number represents a node.

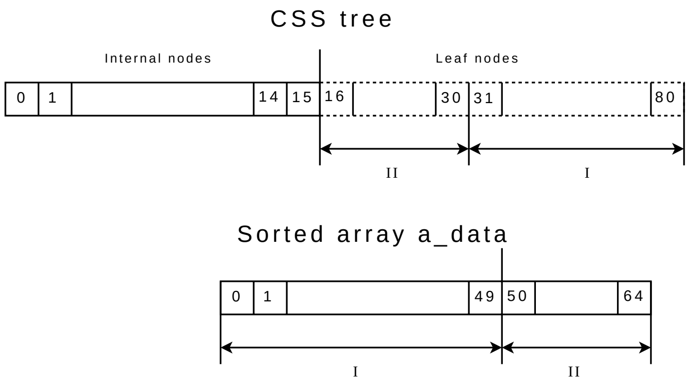

Figure 3.3: A CSS–tree represented as an array.

nodes from left to right, level by level, meaning that nodes 31–80 are read before 16–30. To preserve key order, `a_data` must be divided.

Every node in the CSS–tree contains m keys and has $m+1$ children. Therefore, the children of a node b are numbered as the interval described in Equation 3.1.

$$b(m+1)+1 \qquad \text{to} \qquad b(m+1)+(m+1) \tag{3.1}$$

and the array element at the ith index position is located in node number $\lfloor \frac{i}{m} \rfloor$. As a result, the location of the keys in the children of the node containing key i are calculated by Equation 3.2.

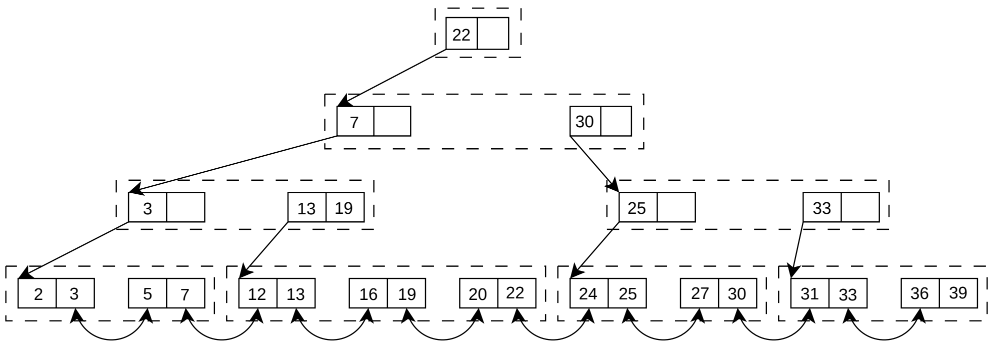

Figure 3.4: Example of a CSB$^+$–Tree of order 1.

$$((m+1)\cdot\lfloor\tfrac{i}{m}\rfloor+1)\cdot m \quad \text{to} \quad ((m+1)\cdot\lfloor\tfrac{i}{m}\rfloor+m+1)\cdot m \tag{3.2}$$

As an example, we use the tree from Figure 3.2. We start in key eight which is located in node two. Using Equation 3.2, the keys in the child nodes (11–15) are then the range from $((4+1)\cdot 2+1)\cdot 4$ to $((4+1)\cdot 2+4+1)\cdot 4$, i.e., 44 to 60. $\lfloor\frac{44}{4}\rfloor = 11$ and $\lfloor\frac{60}{4}\rfloor = 15$, which verifies that the correct keys are found.

While CSS–trees facilitate quick searching, insertions and deletions pose a bigger issue as described in the introduction to this section. However, while incrementally updating a CSS–tree is not possible, it is relatively cheap to rebuild it [25].

CSB$^+$–Tree CSB$^+$–Trees are a mix of B+ and CSS trees, where the goal is to retain the search speed of CSS-trees as much as possible, while better facilitating incremental updates similar to B+ trees. This is done by introducing pointers, but the number of pointers is kept to a minimum in order to achieve as good memory utilization as possible [26].

Figure 3.4 shows a CSB$^+$–tree of order 1. Each dashed box represents a node group, and the arrows represent pointers. Node groups ensure that sibling nodes are stored contiguously in memory. In the example shown in Figure 3.4, a node group can contain at most three nodes.

Insertion and searching is similar to B$^+$–trees. If a leaf node cannot contain the new node to be inserted, it is split in two and the tree is updated as with B$^+$–trees. The major difference is that the old node group of size s is deallocated and a new group of size $s+1$ is allocated, where s is the number of nodes in the group.

CSB$^+$–trees map quite well to GPU memory as sibling nodes are stored contiguously in node groups. The extra space for nodes in the groups means that new memory does not have to be allocated to the index as often as, e.g., B$^+$–trees.

Choise of Index Structure The choice of index boils down to two primary candidates: CSS– and CSB$^+$–trees. CSS–trees have the advantage of faster searching and less space overhead [26], while CSB$^+$–trees facilitate better incremental updating. As the DBMS is intended to be used in an environment dominated by searches and operations on existing data, incremental updates is less of a concern compared to fast execution. Because of this, we choose CSS–trees as the index structure for the DBMS.

3.4.5 Kernel Library

Interface function	**Return value**
PerformOperation(operation, table, index, output table)	void

The kernel library contains the CUDA kernels which are used to perform the database operations supported by the DBMS.

As mentioned in Section 2.4, using a set of computation primitives for parallel operations is a good basis for creating a flexible system capable of a wide range of operations on data in parallel. We base our work on the primitives and their definitions presented in [9] which are:

- Map
- Scatter
- Gather
- Prefix scan
- Split
- Sort

These can then be used to construct a set of operations covering standard database operations as well as mathematical operations. As mentioned in Section 3.3, the database operations should include the basic operations in relational calculus and the mathematical operations should be based on the math functions in .NET. Implementing equivalent versions for all the common static functions defined on the `Math` type in .NET would allow us to translate such function calls to our own version with our query provider.

3.5 Design Decisions

This section summarizes the design decisions made throughout Section 3.4. Most of the choices provide specific requirements for the implementation though a couple of them require more experience to make a final decision.

- We use a custom LINQ provider as query provider for the DBMS.
- SQL is used to represent queries internally in the DBMS. We can optionally choose to expand the SQL representation if certain GPGPU operations can be better represented by an extended syntax.
- The execution of queries are managed from the host, though if it is feasible for all parts of a given query to be executed on the device, we do so.
- We store the data of tables in a collection of arrays, where each field in each entity matches an array. The relation between entities is described by pairs of arrays which match indices from the related entities.
- We use CSS-trees as indices to the tables, which are stored on the device.
- We use computation primitives as the basis for database operations. The operations include the basic operations from relational calculus as well as a representative amount of math operations.

Combined with the requirements from Section 3.3, these make up the high-level specification for the DBMS implementation described in Chapter 4.

Chapter 4

Implementation

In this chapter we describe the implementation of the major components of the DBMS. For how the DBMS is represented on the GPU, we provide an example of the life cycle of a database table. This covers the major obstacles and details regarding each of the more commonly used database operations.

The main purpose of this chapter is to cover the primary obstacles people would have to solve when implementing a usable GPU DBMS and provide our suggestions for how this can be achieved. The aim is for this to provide a solid base from which subsequent research into GPU DBMSs can go.

This chapter reflects what we have actually implemented. For a discussion of the features the DBMS lacks and other potential future improvements, see Chapter 6.

The DBMS is written in C# 3.0, using the CUDA.NET library version 2.1 for interfacing with the GPU, and CUDA 2.1 has been used to implement the GPU code itself. In Appendix B, we give a brief overview of the source code of the DBMS which is located on the enclosed CD.

4.1 LINQ Query Provider

Since the introduction of C# 3.0, LINQ, or Language Integrated Query, has provided a generic, type–safe query model to retrieve data out of arbitrary sources. The benefit of LINQ is that you can query sources from within your C# code without losing the benefits of your development environment such as Intellisense and type checking. By default, LINQ has implementations, or query providers, for a couple of sources including XML documents and relational databases as well as ordinary object collections. To query other sources it becomes necessary to implement custom query providers yourself, and this is how we provide access to our DBMS.

The details of implementing a query provider can be quite technical and complex, but the main principle is simple; as the query is evaluated, an internal representation of the query is built in a format which can be used to extract data from the target source. The most obvious example of this is, perhaps, the creation of a SQL command which can be understood by a DBMS.

The act of creating the query does not result in its immediate evaluation, just a description of the query to be executed, in the form of an `IQueryable` object. This means that the query uses lazy evaluation and can thus be further modified before execution. Only when you turn the result into an `IEnumerable` collection is the query executed, such as when using a `foreach` loop on the query. Listing 4.1 shows an example of a LINQ query and its usage.

```
// Query gets constructed
var query = from person in people
            where person.Name == John
            select person;

// The query is modified to a more narrow result-set
query = from person in query
        where person.Age >= 18
        select person.Name;

// The query is executed, yielding the resulting objects
foreach(var name in query)
{
   Console.Out.Write(name);
}
```

Listing 4.1: LINQ query examples in C#.

To implement our own query provider we implement the `IQueryable` and `QueryProvider` interfaces, of which the latter is where the main work is performed. The main job of this class is to transform LINQ methods into the desired format. When a LINQ query is evaluated, it is first turned into a number of ordinary method calls as the LINQ syntax is merely syntactic sugar. It is not necessary to implement any specific interfaces for a class to work with LINQ, the compiler just looks at the queried object and determines if it has the proper methods defined. This means that if an object defines methods such as `Select`, `SelectMany` and `Where`, it can participate in a LINQ query.

The series of method calls which the query gets turned into are represented as an expression tree describing its structure and components. For example, when you write `where person.Name == "John"` in the LINQ statement, the resulting `IQueryable` object will contain an `Expression` object with a `BinaryExpression` object inside.

When the query is executed, the `Expression` object is parsed and turned into the representation which the data source can understand. This process is the complex part of implementing a LINQ provider and it requires you to consider all the different ways your data source can be queried, including any edge cases there might be. You can include as little or as much intelligence in how this process is handled as needed, depending on how powerful and complete a system you need.

Instead of starting from scratch, we adapt an existing solution to our specific needs. IQ-toolkit is the result of a series of tutorials on the subject of implementing your own LINQ provider and it is designed to be similar to LINQ–to–SQL[27]. While the toolkit is not production quality, it is a good base for building a query provider for a database. The version of the toolkit we use is from the 13th tutorial, so any subsequent updates to the tutorial are not included in our version.

In essence, our query provider does three things for us. The primary function is that it translates the expression into a string which our internal system can consume. This string must include a full description of the work to be performed on the GPU. To this effect, the second function is to capture method calls which we are able to perform on the GPU and include them in this representation. Among the most obvious choices are the functions defined on `System.Math`. The final feature, which the query provider must support, is to create a projection expression which can be compiled into a function used for instantiating the resulting objects and does all the work which is not supported by the DBMS, such as unsupported method calls.

One of the most extensive modifications we have made to the toolkit was for transforming compatible method calls into textual representations — a process we call *internalizing*. For this purpose we created the class `MethodInternalizer` which adds a final pass to the query representation produced by the toolkit. A lot of the work done by this class is complicated by the fact that `Expression` objects are immutable and if you want to modify one, you have to construct a whole new expression tree with the modification.

The job of turning an expression into a query string is done by a visitor class, but before this step, the toolkit removes all parts of the expression not supported by the DBMS and adds them to the projection expression. What the internalizer does is to reverse this process for each function call supported by our DBMS by visiting each node of the projection expression, producing a new version without the method calls being internalized. These method calls are then added to a new expression which replaces the database query expression.

Listing 4.2 shows how the expression visitor replaces an internalizable method with a column declaration. The `ColumnDeclaration` and `TableAlias` classes are used by the toolkit to keep track of the names of tables and to which table a column is associated. If the visited method is not internalizable each argument is visited and a new `MethodCall-Expression` object is returned.

```
public Expression VisitMethodCall(MethodCallExpression methodCall)
{
    if (IsMethodInternalizable(methodCall))
    {
        internalizedMembers = true;
        var columnDeclaration = new ColumnDeclaration(currentName, methodCall);
        columnDeclarations.Add(currentName, columnDeclaration);

        TableAlias alias = TableAlias.Aliases[TableAlias.Aliases.Count - 1];
        return new ColumnExpression(methodCall.Type, tableAlias, currentName);
    }
    else
    {
        List<Expression> args = new List<Expression>();
        foreach (var arg in (methodCall).Arguments)
        {
            args.Add(Visit(arg));
        }
        return Expression.Call(methodCall.Method, args.ToArray());
    }
}
```

Listing 4.2: Internalizing a method call.

Constructing an intermediary string representation of each query is obviously not the most efficient way to execute a query as it requires the DBMS to re–parse the query. However, there are a lot of different cases and conditions when parsing a LINQ query so for simplicity we rely on IQtoolkit to produce SQL output modified to fit our needs. This also provides an added benefit that in many cases it simply becomes a matter of checking the correctness of the query string to debug a query.

4.2 Table Lifecycle

In this section we describe the implementation of common database use patterns throughout the life cycle of a table. Each pattern is introduced by the LINQ query which invokes the behavior from our DBMS. For people with a background in functional programming, the concepts of *select*, *aggregate* and *where* are equivalent to the concepts of *map*, *reduce* and *filter*, respectively.

Before describing the use patterns for the DBMS, we give a brief overview of how queries are executed in general. In the following sections, we use this as a basis for explaining how the different cases are executed.

The central module for handling queries is the `QueryManager` class. It handles the parsing and execution of query strings as shown in Listing 4.3.

```
public QueryResult ExecuteQuery(string queryText)
{
    IEnumerable<string> tokens = queryText.Split(new string[]{" ", Environment.
        NewLine}, StringSplitOptions.RemoveEmptyEntries);
    GpuQuery query = new GpuQuery();
    ParseQueryString(tokens.ElementAt(0), tokens.Skip(1), query);
    QueryResult result = ExecuteQuery(query);
    memoryManager.ClearTemporaryTables();
    return result;
}
```

Listing 4.3: The steps involved in executing a query.

The first step in the process of executing a query is to parse the string representation and turn it into something the DBMS can work with. This is implemented in a functional style, by parsing the command at the head of a stream of command tokens, such as `SELECT` and `WHERE`. Parsing each command also removes the parameters for the command from the stream, thereby leaving the next command at the head. During the process of parsing the different commands, a `GpuQuery` object is passed around and annotated with all the necessary information for executing the query. Listing 4.4 shows how this class is implemented to give an idea of how a query is represented.

```
class GpuQuery
{
    public Dictionary<string,ITable> From { get; set; }
    public string Where { get; set; }
    public List<string> Select { get; set; }
    public GpuQuery ChainedQuery { get; set; }
    public int Top { get; set; }
    public int Skip { get; set; }
    public bool IsDistinct { get; set; }

    public GpuQuery()
    {
        From = new Dictionary<string, ITable>();
        Select = new List<string>();
        Top = 0;
    }
}
```

Listing 4.4: The `GpuQuery` class.

4.2.1 Create Table

Listing 4.5 shows how to create a new database instance and how to create a table on it. The `Table<T>` class implements the `IQueryable` interface and can therefore be queried for data using LINQ. The type argument `T` is the type of objects which are stored in the table as well as the type of objects which can be retrieved from it.

```
var GpgpuDatabase db = new GpgpuDatabase(Console.Out);
Table<Vector> vectorTable = db.CreateTable<Vector>("vectors");
```

Listing 4.5: How to create a new table in the GPU DBMS.

Once the table has been created, it is possible to add an index. This is done by calling the `CreateIndex` method as shown in Listing 4.6, where `"X"` is the column to index.

```
vectorTable.CreateIndex("X");
```

Listing 4.6: Adding an index to the `vectorTable` table

The internal representation of the table is a collection of columns. Each column is essentially a data–parallel array containing all values of a given member on the table type. In isolation, the `Column` class is very similar to the array types in other GPU libraries such as the Accelerator library mentioned in Section 2.4. They encapsulate the fact that data is stored on the GPU and allows for a number of operations to be performed on the elements in the array, examples of which are described in the following sections.

4.2.2 Insertion

Listing 4.7 shows the process of inserting values into a table, a process which mirrors lists or similar collections. Adding new data to a table has been implemented naively, for sake of simplicity, such that each call to the `Insert` method triggers an array allocation on the GPU for each member in the type along with a set of data transfers.

```
vectorTable.Insert(new Vector[]
{
    new Vector(){ X = 3, Y = 9, Z = 9 },
    new Vector(){ X = 3, Y = 0, Z = 0 },
    new Vector(){ X = 5, Y = 9, Z = 9 },
    new Vector(){ X = 3, Y = 0, Z = 4 }
});
```

Listing 4.7: How to insert rows into a table.

4.2.3 Selection

Listing 4.8 shows the most simple way to retrieve data from a table, selecting each record. Listing 4.9 shows the internal textual representation of the query. It illustrates how a selection is mapped to columns on tables.

```
var vectors = from v in vectorTable
              select v;
```

Listing 4.8: How to perform a simple selection on a table.

```
SELECT t0.X, t0.Y, t0.Z
FROM vectors AS t0
```

Listing 4.9: The internal representation of a select query.

As described in Section 4.1, the resulting query object must, in addition to the query string, contain the logic to be performed on the host after the query has been executed. In this case, the logic is a function which instantiates a new `Vector` object with the values `X`, `Y` and `Z`.

By default, the IQtoolkit would try to apply a value to each property of the object, but this was changed such that it only applies to properties which are not read–only. For the `Vector` class, we added a `Length` property which can be derived from the vector coordinates instead of being stored as a value. Other such changes can be made to give a better control over how a type is treated in the DBMS such as custom attributes for annotating members and properties.

For selections, the `GpuQuery` object is annotated with a number of strings, one for each member in the output type. Listing 4.10 shows how such a string can look.

```
ADD(t0.X, POW(t0.X, 2))
```

Listing 4.10: A selection string.

As shown, each operation on the selection is represented by a function name. These functions are recursively evaluated, with each evaluation resulting in a temporary column. Taking the `ADD` function as an example, this would result in a binary function being performed as shown in Listing 4.11.

```
case "ADD":
    pair = GetBinaryParameters(parameters);
    return BinaryOperation.ExecuteBinaryFunction(
        BinaryFunction.Add,
        ExecuteSelect(pair.A, from),
        ExecuteSelect(pair.B, from)
    );
```

Listing 4.11: The addition of two columns,

The static method `BinaryOperation.ExecuteBinaryFunction` handles the job of invoking the proper kernel on the GPU as shown in Listing 4.12. The `Execution` class is a helper method which wraps the process of invoking a kernel using CUDA.NET. Once the kernel has been executed, the result is wrapped in a `Column` object and returned. If either of the two input columns are temporary they are disposed which makes sure that temporary data has as short a lifespan as possible but it still requires GPU memory of several times the size of the original data. For large tables, this would necessitate a significant amount of GPU memory reserved for temporary data.

```
public static Column ExecuteBinaryFunction(BinaryFunction function, Column a,
    Column b)
{
   //Select function name (op) based on function and column types

   Execution exec = new Execution("math", op);
   int size = Math.Max(a.Size, b.Size);
   if (a.IsConstant) exec.AddParameterValue(float.Parse(a.ConstantValue));
```

```
    else exec.AddParameterPointer(a.Pointer, size);
    if (b.IsConstant) exec.AddParameterValue(float.Parse(b.ConstantValue));
    else exec.AddParameterPointer(b.Pointer, size);

    CUdeviceptr result = exec.AllocAndAddParameterPointer<float>(size);
    exec.AddParameterValue((uint)size);

    exec.Execute();

    FreeTemporaryColumns(a, b);
    return new Column(typeof(float), result, size);
}
```

Listing 4.12: The execution of a kernel from C#.

4.2.4 Database Selection Functions

In addition to the relational calculus which might be seen as the main way to interact with a database, the DBMS is also designed to be able to perform a number of common functions on the GPU. Listing 4.13 shows how math functions can be used in a query where these functions are then executed on the GPU. Both ordinary math operators such as addition and functions such as square root are turned into GPU versions. The math functions which the DBMS currently support are the standard operators of addition, subtraction, multiplication and division as well as the functions for square root and exponentiation. Adding other functions where each value can be calculated on its own thread and for which the CUDA library contains a matching function is trivial. For a complete list of such CUDA functions, see [1].

```
var query = from v in vectorTable
            let length = (float)Math.Sqrt(Math.Pow(v.X, 2) + Math.Pow(v.Y, 2) +
                Math.Pow(v.Z, 2))
            select new Vector() { X = v.X / 2, Y = (float)Math.Pow(2, v.Y), Z =
                length };
```

Listing 4.13: A database query using on–device execution of math functions.

Listing 4.14 shows the textual representation of the query in Listing 4.13 after the math functions have been internalized as described in Section 4.1.

```
SELECT DIV(t0.X,2), POWER(2,t0.Y), SQRT(ADD(ADD(POWER(t0.X,2), POWER(t0.Y,2)),
    POWER(t0.Z,2)))
FROM vectors AS t0
```

Listing 4.14: The textual representation of a database query, including several math operations.

Another type of function on collections is aggregations such as sum and average. These operations already exist as extension methods for `IEnumerable` objects, and by extension, IQueryable objects. They result in a single value and can be used as shown in Listing 4.15. The aggregations currently supported by the DBMS are min, max, average and sum.

```
var sumX = vectorTable.Sum(vec => vec.X);
var minX = vectorTable.Min(vec => vec.X);
var maxX = vectorTable.Max(vec => vec.X);
var avgX = vectorTable.Average(vec => vec.X);
```

Listing 4.15: Constructing a query using aggregates.

In contrast to many of the computations we perform on table columns, aggregate functions do not have a single operation on each column element. Aggregate functions collect a result from all the elements in the column, gathering the results using some combination function such as addition for the summation aggregation. This pattern needs intermediary results which must be read by different threads and as such has higher requirements for synchronization and optimization.

Listing 4.16 shows the aggregation kernel. The parameter func signifies which aggregation function to combine the result with. The apply_func method switches on this value and invokes the proper aggregation function. We have used this principle of developing generic versions of different kernels to a high degree, in order to simplify the development of CUDA code.

```
__device__ void
aggregate(float *g_idata, float *g_odata, unsigned int n, unsigned int func)
{
   __shared__ float sdata[512];
   unsigned int tid = blockIdx.y * gridDim.x * blockDim.x * blockDim.y;
   tid += blockIdx.x * blockDim.x * blockDim.y + threadIdx.x + threadIdx.y *
       blockDim.x;

   if(tid < n)
      sdata[tid] = g_idata[tid];
   __syncthreads();

   for (unsigned int s = blockDim.x; s > 32; s >>= 1)
   {
      if (tid < s && tid + s < n)
      {
         sdata[tid] = apply_func(sdata[tid], sdata[tid + s], func);
      }
      __syncthreads();
   }

   if (tid < 32)
   {
      sdata[tid] += sdata[tid + 32];
      sdata[tid] += sdata[tid + 16];
      sdata[tid] += sdata[tid +  8];
      sdata[tid] += sdata[tid +  4];
      sdata[tid] += sdata[tid +  2];
      sdata[tid] += sdata[tid +  1];
29      }
30
31      if (tid == 0) g_odata[blockIdx.x] = sdata[0];
32  }
```

Listing 4.16: Aggregation CUDA kernel.

Line 4 in Listing 4.16 shows the allocation of an array of per–block shared memory. This provides a fast temporary storage in which to aggregate the values. For each iteration of the for loop in the algorithm, the two halves of the remaining data set are aggregated into the first half, each thread working on two values. Aggregations are not perfectly suited for GPU execution as each iteration of the algorithm reduces the amount of active threads in half. This illustrates a case where an algorithm differs particularly between CPU and GPU implementations as the aggregation algorithm is linear on the CPU but has a complexity of *O(n log(N))* on the GPU.

Since more than one thread accesses the same indices in the shared memory, we have to synchronize between each iteration. Synchronizing threads on a GPU is relatively cheap compared to synchronizing on the CPU as the barrier is implemented in hardware.

Note that the implementation of the aggregation function in Listing 4.16 is not optimized as much as possible. NVIDIA has used the sum reduction as a case study in how you can optimize different aspects of CUDA programming through a series of seven steps[28]. As we are not using C++, we cannot take advantage of the template–based optimizations as they are resolved at compile time of the kernels but it illustrates the type of considerations we would have to make for optimizing different types of kernels.

An optimization we have implemented is to take advantage of the fact that if no more than a warp, or 32 threads, are executing in a block, they do so in lock–step. This means that we do not have to synchronize between iterations in lines 21 to 29. Optimizations such as these can be difficult to identify and reduce the readability of the code, and it is a good illustration of why many developers would prefer working at a higher level of abstraction, such as the one provided by our DBMS, even at the cost of some performance.

4.2.5 Where

Listing 4.17 shows how to construct a query which filters the results.

```
var vectors = from v in vectorTable
              where v.X == 1 && v.Y < 6
              select v;
```

Listing 4.17: Filtering the result of a query using a 'where' clause.

The principle when filtering the result is similar to selections in that a number of functions are evaluated, only the values are now boolean instead of numeric. Listing 4.18 shows how the query from Listing 4.17 looks to the DBMS.

```
SELECT t0.X, t0.Y, t0.Z
FROM vectors AS t0
WHERE AND(EQ(t0.X,1),LT(t0.Y,6))
```

Listing 4.18: Internal representation of a filtering query.

When evaluating the last line of the query, the same recursive function evaluator mentioned in Section 4.2.3 is used. The result of each step in the calculations is still a column, only now containing boolean values. The final result of the filter clause is a column containing 1's for each record matching the filter and 0's for the records which fell outside of the filter.

If the `X`–column is indexed in the example in Listing 4.17, the column is searched using a modified version of the search algorithm in [25] instead of applying a boolean function to each element. This returns a list of 1's and 0's just as the non–indexed comparison of the `Y`–values. Section 4.2.5 describes the index in greater detail.

The final step in applying the filter is to transform the source tables into a version only containing the matched records. As all the table data is stored on the GPU, this is done through a number of kernels. Listing 4.19 shows the pseudo code for this procedure.

```
1  columnOfIndices = Evaluate(filterExpression)
2  numberOfMatches = columnOfIndices.Sum()
3  condensedIndexArray = allocateGpuMemory(numberOfMatches)
4  condenseIndexArray(columnOfIndices, condensedIndexArray)
5  foreach column in query.From
6  {
7      columns.Add(column.Gather(condensedIndexArray))
8  }
9  query.From = new Table(columns)
```

Listing 4.19: Pseudo code for generating filtered tables.

The main work in this process is to construct an array containing the indices into a table which matches the filter. To illustrate, we use the example column shown in Figure 4.1 (a) with the filter $n > 2$. After evaluating the filter expression in the first line of Listing 4.19, we have a column with 1's and 0's, representing matches as shown in column (b). In line 2, we use the sum aggregate function on this column to determine the number of matching rows, a value which is used to allocate space to hold an index array with the affected indices and which is six for our example.

The purpose of line 4 is to populate the newly allocated array with the indices of the rows matched by the predicate using a two–step process. First, the prefix sum of the result of the filter operation, i.e. Figure 4.2 (a), is calculated, resulting in column (b). Second, a kernel is invoked to turn this column (b) into a column of matching index values, filling the column allocated in line 3. The resulting column, (c), is created by the kernel shown in Listing 4.20.

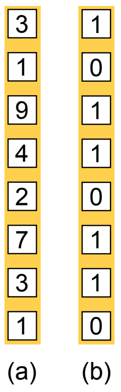

Figure 4.1: The input column (a), filtered with the predicate $n > 2$ in (b).

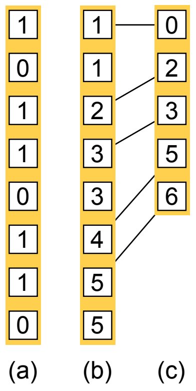

Figure 4.2: The filter result (a), with prefix sum applied in (b) and an index column in (c).

The kernel is executed with a number of threads equal to the original number of columns, i.e. eight in this example, and each thread looks at one value and the value before it in column (b). If the values are different it means that there was a match and the thread has to write a value to the output column. The value in (b) minus one equals the index in column (c) to write to and the thread id is the value to write.

```
extern "C"
__global__ void
definePredicateIndexArray(float *predicatesPS, float *g_o_indices, unsigned int
    n)
{
   unsigned int tid = blockIdx.y * gridDim.x * blockDim.x * blockDim.y;
   tid += blockIdx.x * blockDim.x * blockDim.y + threadIdx.x + threadIdx.y *
       blockDim.x;

   if (tid < n)
   {
      int curr = (int)predicatesPS[tid];
      int prev = (int)predicatesPS[tid - 1];
      if (tid == 0) prev = 0;
      if (curr > prev) g_o_indices[curr - 1] = tid;
```

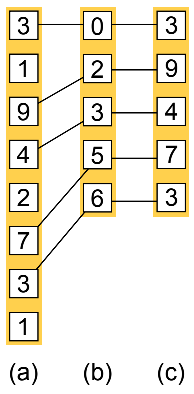

Figure 4.3: Using an array of indices (b) to gather certain values from a large column (a) into a smaller coulmn (c).

```
    }
}
```

Listing 4.20: A kernel for finding the index values of filter matches from an prefix sum column.

Given that we now have a list of the indices at which matches where found, all that is left is to create a temporary table, where the matches are copied to. In lines 5 to 8 in Listing 4.19, a new column is created for the temporary table by applying the gather function to the original column using the index column we have generated. Figure 4.3 shows how the index values in column (b) are used to copy selected values from the original column (a) into the new column (c). Listing 4.21 shows the kernel used to apply the gather operation.

```
extern "C"
__global__ void
gather(float *g_idata, float *g_indices, float *g_odata, unsigned int n)
{
   unsigned int tid = blockIdx.y * gridDim.x * blockDim.x * blockDim.y;
   tid += blockIdx.x * blockDim.x * blockDim.y + threadIdx.x + threadIdx.y *
       blockDim.x;

   if(tid < n)
   {
      int idx = (float)g_indices[tid];
      g_odata[tid] = g_idata[idx];
   }
}
```

Listing 4.21: The *gather* parallel primitive.

Index

The size of each node in the CSS–tree is set to 512 values, the same size os the block size which searches the tree. This allows warps to optimize their memory throughput using coalesced accesses and share data quickly with shared memory.

When a block searches a node, each thread looks at the element matching its id and the adjacent one to determine where to navigate next in the tree. Using the shared memory, the threads broadcasts the results and takes the left–most match. This process continues until the leaf nodes are reached. Depending on the type of search, the algorithm moves right or left searching for the bounds of the matches. The algorithm makes sure that the left–most value is the one found and if the filter function is *equals*, the thread block shifts to the right until the filter is no longer true. Similarly for the other comparison operators, the block has to find the value which ends the range of matches.

The above process is performed by a single block which does not provide an optimal use of the hardware. However, the implementation allows several blocks to be executed simultaneously by searching for different values. This allows the index to evaluate several filter arguments at once.

4.2.6 Deletions

Methods for deleting tables and rows are not included in the LINQ interface and as such are just ordinary functions on the `Table` object. Save for the deletion of a subset of table rows, the operations are not complex. Deleting a table requires it to be unregistered from the `MemoryManager` as well as emptying each of its columns. Deleting all the rows in a table just requires the pointer to the GPU memory to be freed.

We have not implemented the deletion of individual rows but have provided the method signature as shown in Listing 4.22 which illustrates how selective deletions would occur. Implementing this function would be similar to evaluating a ‘where’ clause to identify the indices of rows matching a predicate.

```
public void Remove(Predicate<T> where)
{
   //Evaluate query
}
```

Listing 4.22: The signature for table row deletions.

Chapter 5

Performance Measuring

In this chapter we describe the tests performed on the DBMS. We execute a series of operations on our DBMS as well as a Microsoft SQL Server 2008 database. We then compare the results and briefly describe the outcome of the tests.

5.1 Method

We test the performance of our DBMS to provide an indication of how fast it is for performing common tasks, with a focus on the its primary use cases. This includes inserting large quantities of data in bulk, performing aggregate operations such as sum and average, as well as searches. Insertions and selections are performed on both indexed and non–indexed columns to see how this affects performance. We perform the tests on data of several different sizes to show how performance scales as the database grows. We also test several complex math operations on the data, testing the different functions supported by the DBMS. The main purpose of this test is to determine if the DBMS has introduced overhead or other issues which prevents it from taking full advantage of the computation power of the GPU.

All tests are performed on a computer with the following specifications.

- Intel Pentium Dual–Core 2.5GHz with 800MHz FSB
- 4GB DDR2 400MHz RAM

- Abit I–G31 Motherboard
- GeForce 8600GTS 256MB
- Maxtor DiamondMax 500GB SATA 7200RPM *Hard Disk Drive* (HDD)

The operating system used is Microsoft Windows XP SP3 32bit, so only 3.25GB system memory is available. The GPU uses the PCI–Express bus to transfer data, which means that the theoretical bandwidth from system memory to GPU is 4GB/sec. The database is used out–of–the–box with a single table named 'Vectors' which contains an auto–incremented, unique integer identifier and three attributes of type `float`. No optimizations have been made to the database to increase performance. The initial size of the database is set to be large enough to avoid incremental resizing during the tests.

Each test is performed five times, of which the minimum and maximum results are excluded, to remove outliers, and the remaining three are used to calculate the average run time of the test. Execution time of code on the GPU includes any time it takes to transfer data between the GPU and main memory to provide a more realistic scenario.

Rather than using loops in the code to execute a given method several times, we use an automated script that runs the test executable a number of times with the proper parameters. We do this to avoid issues with garbage collection and ensure homogeneous conditions for each test. Also, after each test is completed, the databases are purged of all data.

5.2 Insertion

The first thing we test is the time it takes for inserting elements into the databases. To do this, we insert arrays from 50,000 to 500,000 vectors in steps of 50,000. Each vector consists of three float values. We do this for both our database and the SQL Server database and compare the time spent. This is done both using indexed and non–indexed data in order to determine the cost of maintaining the indices.

The results for insertions can be seen on Figures 5.1 and 5.2 for SQL Server and our DBMS respectively. Timing on the y–axis is measured in milliseconds and the number of elements on the x–axis is measured in thousands.

We observe that the time to insert a given number of elements into our database is roughly a factor 100 faster than the SQL Server database. The insertion time scales linearly in both cases, which matches our expectations as any overhead with a single bulk insertion is constant or at most linear in the number of values.

It was also expected that insertions into the SQL Server database would be much slower as the differences in bandwidth to GPU memory and the HDD is substantial. Also, the SQL

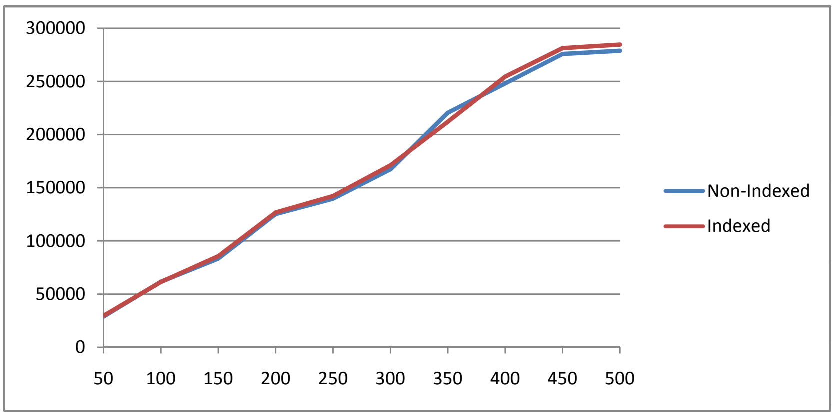

Figure 5.1: The insertion time (y–axis) of n–thousand values (x–axis) into SQL Server.

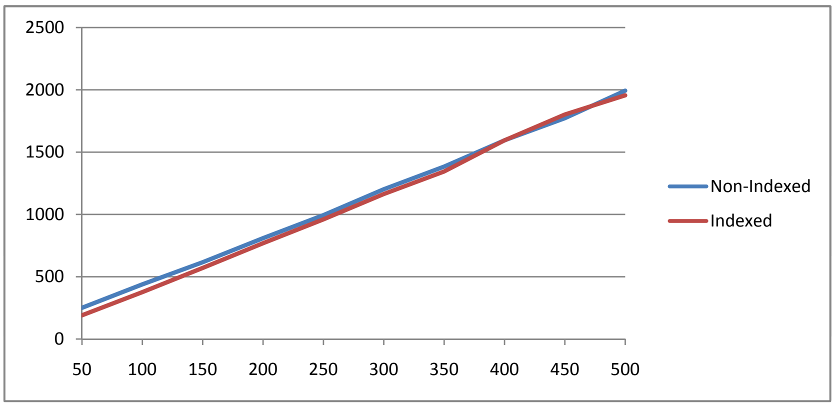

Figure 5.2: The insertion time (y–axis) of n–thousand values (x–axis) into our DBMS.

Server database supports functionality such as logging and roll–backs of transactions. This induces additional overhead compared to our DBMS which does not have similar features. The reason we have not disabled these features on the SQL Server is that we wish to test the systems in their most likely configuration rather than setting up some theoretical but unrealistic scenario.

Included in the test is also insertion while maintaining an index on one of the `Columns`. We observe that the cost of building the CSS–tree is indeed very low as suggested in [25]. Maintaining an index on the SQL Server database is also very low–cost, which is substantiated by the tests which show almost no increase in execution time.

To test how expensive incremental insertions are, we insert a total of 250,000 vectors in batches over 2, 5, and 10 iterations. Appending in the SQL Server database takes roughly the same amount of time no matter how many times the table size increases. However, we observe that incrementally appending data to tables using our database takes increasingly more time as the number of iterations rises as shown in Figure 5.3. In the figure, the x–axis is the number of milliseconds the total insertion took.

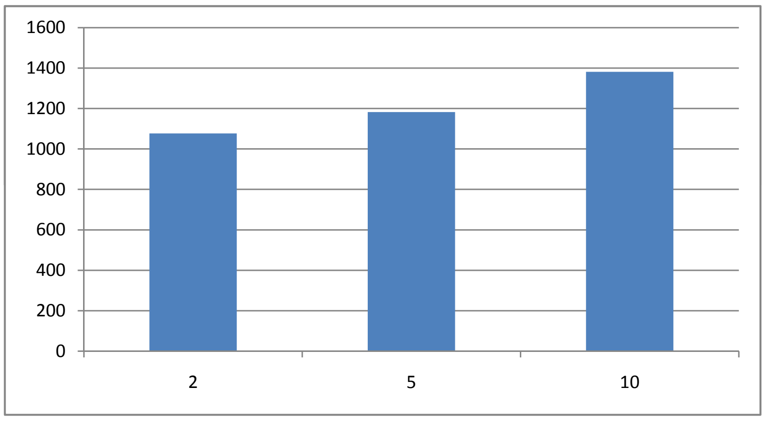

Figure 5.3: The time in milliseconds to append 250,000 vectors over 2, 5, and 10 iterations.

The reason for the decreasing insertion rate is the design described in Section 3.4.3. Each time data is added to a table, the old table is copied to a new table with room for the new data, which is then inserted. As the table grows, more and more data must be copied with each insertion, which causes the slowdown.

For completeness, we have included the time it takes for SQL Server to delete rows, the result of which can be seen in Figure 5.4. The figure shows that deletion times are even longer than insertion times and that there remains a slight overhead in managing the index. Deletion is near–instant on our DBMS as it involves little more than freeing a the pointers referencing the GPU arrays with the data. As with insertions, this reflects the fact that the SQL Server has to erase the data from disk and maintain certain integrity checks.

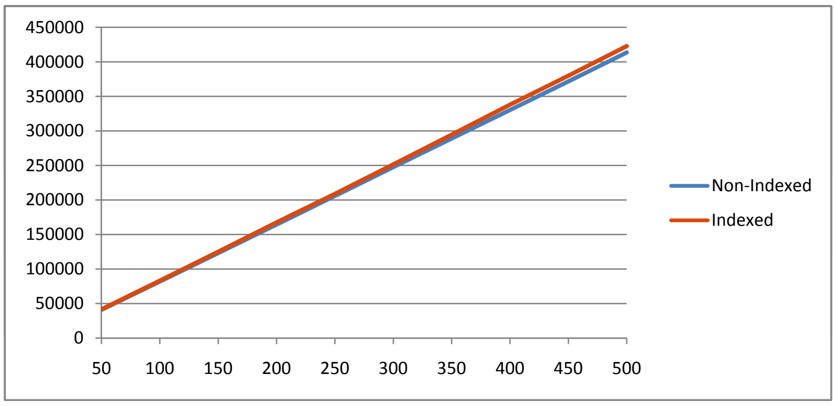

Figure 5.4: The deletion time (y–axis) of n–thousand values (x–axis) from SQL Server.

Due to the very slow insertion and deletion rate of SQL Server, we expect that there are ways in which this process can be sped up a great deal, based on the level of data integrity required.

5.3 Selection

This test measures the time it takes for selecting all elements in the databases. To do this, we select from 50,000 to 500,000 vectors in steps of 50,000. We do this for both our database and the SQL Server database and compare the time spent.

The result of the test is shown in Figure 5.5 with time spent on the y–axis an the thousands of values on the x–axis. Compared to SQL Server which scales perfectly linear, our DBMS suggests a more exponential growth. This result did not match our expectations and could prove very problematic for the performance of the DBMS.

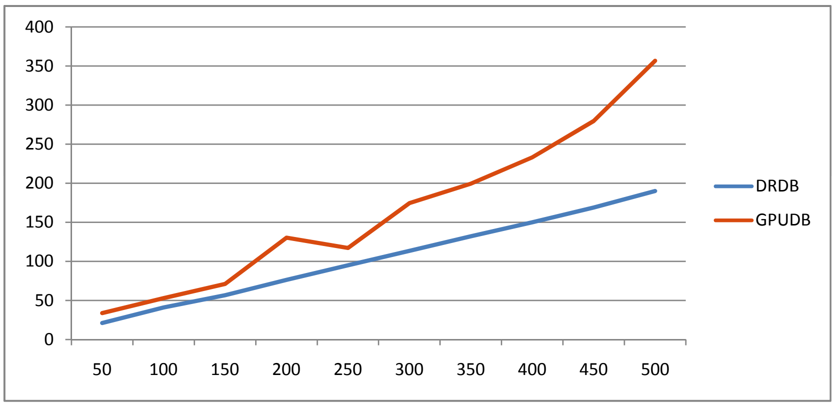

Figure 5.5: The selection time in milliseconds (y–axis) of a complete table with n–thousand values (x–axis).

It is difficult to see from the test whether growth is linear or exponential in our DBMS, while the SQL Server database is decidedly linear. In order to determine how selections on our DBMS scales, we perform a second test on larger data sizes. The results of this test is shown in Figure 5.6.

As the figure shows, the revised test shows that our DBMS indeed grows linearly with the data size.

Due to problems getting some of our kernels to scale beyond a single block, we have been unable to perform meaningful tests on selections of specific values. This prevents us from testing predicate filters properly, with and without indices. However, to get an idea of how well our index works, we have tested it in isolation. The test measures the time it takes to select 1,000 elements in the middle of a table containing from 50,000 to 500,000 vectors. We compare this to the time it takes SQL Server to select 1,000 elements from an indexed table. The result of the test is shown i Figure 5.7.

The test shows that our index performs well compared to the SQL Server database index, although it must be repeated that it is not a fair comparison as less work is done by our index. The main purpose is to determine how it scales. The SQL Server database results

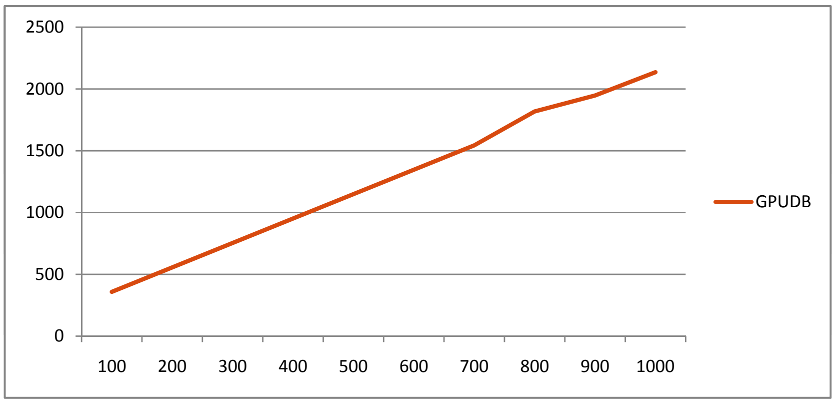

Figure 5.6: The selection time in milliseconds (y–axis) of a complete table with n–thousand values (x–axis).

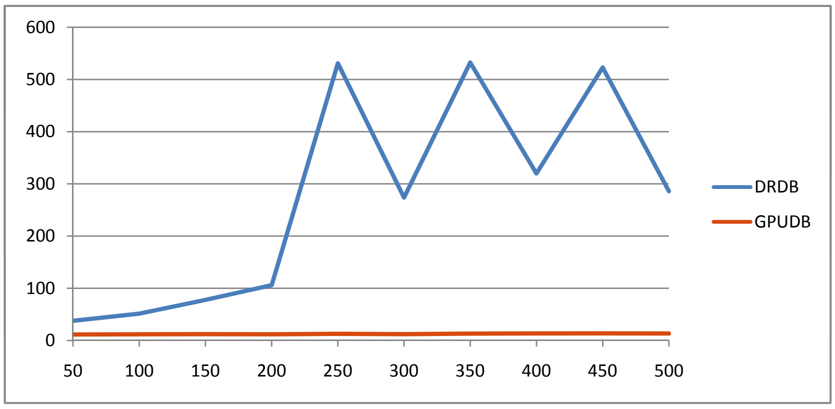

Figure 5.7: The selection time in milliseconds (y–axis) of 1,000 indexed elements with n–thousand values (x–axis).

have a very high variance in time, from test to test. We have been unable to determine what causes this behavior, but the results still indicate that it scales linearly.

It is more difficult to determine how our index scales as the results are dominated by overhead from, e.g., kernel invocations. Therefore, we test our index on arrays of size 1,000,000 to 8,000,000 in one million increments. The results of this test are shown in Figure 5.8.

This test shows a linear increase in time as the number of elements increase. The previous test does not indicate that a direct comparison with the SQL Server database would be meaningful. Also, as mentioned, the workloads are very different, so any comparison with this much data would not be realistic.

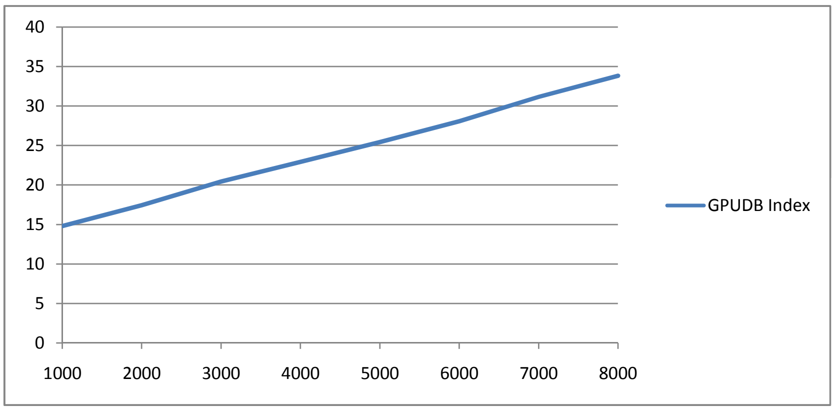

Figure 5.8: The selection time in milliseconds (y–axis) of 1,000 indexed elements with n–thousand values (x–axis).

5.4 Math Operations

In this test we execute a complex math operation on arrays of between 50,000 and 500,000 vectors on both databases. The specific type of math operations should not be relevant as they all reflect the computation power of the processor.

Figure 5.9 shows the result of the test and it is clearly visible that the GPU is highly suited for this task. This test performs as expected though there could potentially be room for much higher performance, as other parts of the DBMS might be providing a bottleneck. The SQL Server generally performs about a factor 4 slower than our DBMS.

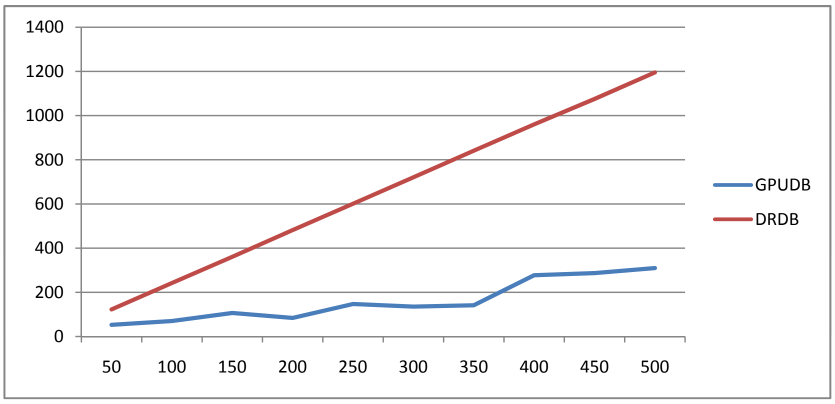

Figure 5.9: The calculation time in milliseconds (y–axis) to evaluate n–thousand (x–axis) math operations.

Chapter 6

Future Work

In this chapter we cover a number of features which are important for a DBMS but which we did not have time to implement. Based on our experiences with the rest of the DBMS, we discuss how we would go about implementing them and what difficulties there might be with making it work on the GPU.

6.1 Data Creation and Transfer

For the DBMS to be more useful there are a number of improvements one could make that would improve the flexibility of interacting with the DBMS. One such case is when you would like to store the result of a computation in a more permanent manner. Listing 6.1 shows how the generic method `ToTable` solidifies the result as a new table. Currently, the only alternative is to store results in objects on the host and then possibly inserting them into a new table.

```
var q = (from p in people
         select p.Name).ToTable<string>();
```

Listing 6.1: Creating a new table from the result of a LINQ query.

Another similar case is that if you know you will no longer need the original input data for some operation you can write the result of the operation to the source columns. Listing 6.2 shows an example where a table of people is overwritten with new Person instances with double the age, saving the memory otherwise required for the temporary result.

```
var q = (from p in people
         select new Person(){ Name = p.Name, Age = p.Age * 2 }).InPlace();
```

Listing 6.2: Change table in place.

The general pattern of the above solutions match existing methods such as `ToArray` and `ToList` so should be familiar to developers. Both the solutions deal with inserting new data into the database but only insofar as it already contains data. It would also be very useful to generate data directly on the database to avoid the transfer overhead.

Listing 6.3 shows how we can define streams of numbers. Using these streams we can construct a table which is instantiated with data from them. As an example, Listing 6.4 shows how we can create a table of integers containing the first 1000 even values. The implementation of the number streams is not relevant for GPU execution as we simply look for their presence when evaluating the LINQ query. If the `Take` method is not called, the DBMS must throw an exception as it makes no sense creating a table from an infinite stream of numbers.

```
public class Numbers
{
   public IEnumerable<int> Even(IEnumerable<int> numbers)
   {
      foreach(var num in numbers)
         if (i % 2 == 0) yield return num;
   }

   public IEnumerable<int> Odd(IEnumerable<int> numbers)
   {
      foreach(var num in numbers)
         if (i % 2 == 1) yield return num;
   }

   public IEnumerable<int> Natural
   {
      get
      {
         for(int i = 0; ;i++)
            yield return i;
      }
   }
}
```

Listing 6.3: Streams of numbers.

```
var numbers = (from n in Numbers.Even(Numbers.Natural)
               select n).Take(1000).ToTable<int>();
```

Listing 6.4: Using number streams to initialize table data.

The resulting query string from performing the query in Listing 6.4 could look as shown in Listing 6.5. The `EVEN(NATURAL())` statement would be mapped to kernels for populating arrays and the `SOLIDIFY` statement would indicate that the result would be stored in a permanent table similar to the `AsTable<T>` function.

```
SELECT t0.p0 TOP 1000
FROM EVEN(NATURAL()) AS t0
```

```
3 SOLIDIFY
```

Listing 6.5: Internal query string for table initialization.

These examples show a couple of ways to generate data on the GPU but are by no means exhaustive as, e.g., other streams and types could be used.

So far this section has dealt with the insertion of data into the database but without addressing the source of the data. As the database lives in volatile storage it is important to consider how to get data into the database and out of it again before closing it. There are generally three scenarios to consider; inserting data through the host program as we have been doing in our current implementation, serializing to and from files and transferring data between different databases.

Serializing data to and from files would require us to write a textual representation of any database meta–data, such as table structures, and just dump all the column data into the file in a comma–separated format or something similar. More interesting, and perhaps more useful, would be to transfer data between the GPU–based database and ordinary databases. This would enable such scenarios as filling up the database with a lot of data from an DRDB, perform a number of calculations on them, update the data with the new information and transfer all the data back to the DRDB. To do this we could, for example, use the FreeTDS library, which is an open source library for establishing native connections to Sybase and SQL Server databases using the Tabular Data Stream protocol [29]. This approach does require us to more or less handle each DBMS specifically, rather than providing a single catch–all solution.

6.2 Futures

Lazy evaluation of queries has one drawback, though, as we are unable to take advantage of the time between constructing the query and needing the results to begin executing the query. As most of the query runs on the GPU, executing the query immediately has little impact on the rest of the program. If the user knows that he is not going to do anything with the query, later, which would facilitate any optimizations, he should be able to initiate an asynchronous execution of the query. In [7] we implemented a feature similar to *futures* where the result of an operation gets executed asynchronously and only blocks to perform them when the result is needed.

There are several approaches to achieving this kind of functionality, one would be to create an extension method on `IQueryable` objects which would start executing the query and return an object representing the future. When you need the result, you would either be able to just use it or the future would block until having completed its work. Listing 6.6 shows how this could look.

```
var future = (from p in people
              select p.Age).Asynch();

//Some work

int[] ages = future.ToArray();
```

Listing 6.6: Asynchronous evaluation of queries.

Alternatively, you could provide a callback method which would handle the results. If needed, you could also stream data to the callback method in bundles of a given size, if immediate feedback was required. Listing 6.7 shows how these two features could be used.

```
var future1 = (from p in people
               select p.Age).Asynch(age => Console.WriteLine(age));

var future2 = (from p in people
               select p.Age).Asynch(1000, age => Console.WriteLine(age));
```

Listing 6.7: Asynchronous evaluation of queries using callback methods.

6.3 Strings

Strings are an important part of almost any type of DBMS and would be a necessary addition to our DBMS. As strings in C are represented as arrays of chars, they pose a challenge to implement. Previous work shows that efficient representations of strings can be achieved[30].

One way to represent the variable–length char arrays would be to define a fixed length as a part of the table declaration, for example using attributes for annotating the table. This would allow a column of strings to be allocated as a single, large array. The fixed lengths would make it possible to calculate the offset to access a given string. Here, a good access pattern is a single thread per sub–string. Alternatively, columns with longer pieces of text could be represented as a jagged array. Here, a better access pattern would be for all the threads to work on a single string at a time, each working on its own part of the string. Having each thread working on its own string would result in an uneven distribution of work unless each string was of the same length — a case that would favor the first allocation model. Both models have their uses just as normal DBMSs have similar representations, such as `varchar` and `blob` for fixed–length and dynamic sized text, respectively. In order for strings to work with indexing, the best way to represent strings would be as a jagged array, sorted by the ASCII value of the characters.

6.4 Index

The index is also a candidate for future improvements. While the CSS–tree functions well for our use case, it would be beneficial to include the option to select other index structures in order to improve DBMS versatility. One such structure could be a CSB$^+$ tree mentioned in Section 3.4.4. The improved functionality for handling incremental updates would be a good way of augmenting the index. Other structures such as KD–trees could also be used to include support for spatial data. The user of the DBMS could then select the appropriate data structure based on the specific use case. This is already used in, e.g., the MySQL DBMS which supports B–trees, R–trees, and hash indices [31].

In order for it to be indexed, a column must first be sorted ascending. Due to time constraints, we currently manually ensure that data is sorted accordingly before being inserted. In order to the sorting automatically on the GPU, a number of sorting algorithms that run on the GPU have been implemented by others such as Bitonic Sort [9], QuickSort [32] and MergeSort [33].

6.5 Energy–Efficient Computing

In 2005, the total power used by servers made up an estimated 1.2% of total U.S. electricity consumption when including cooling and auxiliary infrastructure [34]. In these days where energy efficiency is in focus, it is desirable to be able to reduce the power budget, as well as the electricity bill.

Comparing the power consumption of CPUs and GPUs in relation to their computation power, an Intel Core i7 3.2GHz produces 51 GFLOPS at a peak power consumption of 130W, whereas a GeForce 295GTX produces 1788 GLOPS at a peak of 289W [2, 35]. This makes the GeForce more than 15 times as energy efficient as the CPU when considering GFLOPS per watt at peak performance. As it is very difficult to create an application that utilizes the GPU 100%, the difference is smaller in practice. Still, it shows that there is great potential for creating a more energy–efficient DBMS or query processor using the GPU.

This concept of energy–efficient computing has also been used to create SPRAT, which is a framework for runtime processor selection based on energy–efficiency [36]. The main problem with this is that for it to work, it must be possible to predict the execution time of an application on both the CPU and GPU. In SPRAT, they run the application a number of times to record execution times in order to more effectively select the proper processor. In the context of databases, a query processor using the GPU could use heuristics from previous queries to predict when to perform queries on the CPU and when to use the GPU. This could also include an option to select whether to use the GPU to achieve maximum energy–efficiency or maximum performance.

6.6 Optimizations

Performance is an important aspect of DBMS design and in this section we discuss several categories of optimizations which can be performed to improve the DBMS.

6.6.1 LINQ Queries

A lot of work goes on when evaluating a LINQ query — more so even, with the optimizations mentioned in Section 6.6.2 — so to decrease the amount of work when executing the same query repeatedly, the database can maintain a cache of queries. To determine the equality of two LINQ expressions we could apply the visitor pattern to traverse each nested sub-expression while checking their individual equality. In addition to caching the query itself, the database could also save the `GpuQuery` object which is created by the `QueryManager` so that the query string representation does not have to be re-parsed.

As mentioned in Section 4.1, generating a string representation of the LINQ query and then re-parsing it is not the most efficient way to execute database queries. The textual query representation is useful for debugging but for a production level DBMS we should develop a custom query provider implementation rather than using the IQtoolkit. The toolkit, being the result of an evolving proof of concept, is not designed with performance in mind. As such, it does a lot of generally redundant work, as well as a lot of work which might not be applicable to our DBMS. Developing a new query provider from the bottom up would both allow us to optimize its performance for our specific implementation and directly create a representation of the query which the DBMS understands.

6.6.2 Query Restructuring

One type of optimizations deal with the restructuring of the work which is performed during a query. The concept is to analyze the computation graph and determine if there is any work you can avoid performing because it is either redundant or in other ways do not add any new information to the result. This approach, and a number of the optimizations described in this section are inspired by the optimizations performed on the Accelerator framework described in [15]. The place these optimizations would be introduced is in the LINQ query provider, when constructing the internal representation of the query.

One way to restructure expressions is to look for mathematical operations which can be reordered, for example if one operation is distributive over another, as illustrated in Figure 6.1. Here, we reduce the number of reductions from three to two, by adding the two columns X and Y before multiplying them by two. This both reduces the amount of math operations required as well as the number of memory fetches required to fetch the operation arguments.

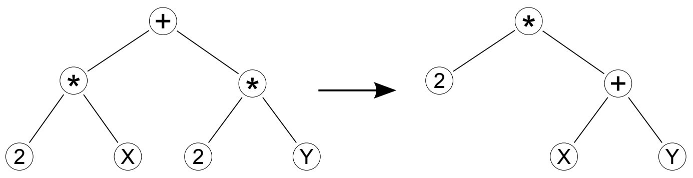

Figure 6.1: Restructuring a mathematical expression to an equivalent version with 2 rather than 3 operations.

We can also restructure boolean expressions to, for example, exploit short circuiting semantics. If we have the expression "X < 5 && Y == 4", any row not satisfying the first predicate could never satisfy the entire predicate. Currently, the database evaluates the two partial expressions first, combining them in the end with the '&&' operator. Instead, we could evaluate the first partial expression, generate the resulting temporary table and then evaluate the second partial expression against this. If there are few values matching the first predicate we can effectively halve the number of operations. Taking this idea a step further, we can exploit the commutative nature of the '&&' operator and evaluate the predicate least likely to result in a large result-set, first. As 'equals' is more restrictive than, for example, 'greater than', we could rearrange the above expression to "Y == 4 && X < 5". Due to the reliance on short circuiting, this does not work directly with the '||' operator, though the method can be adapted using the inverse result of the first predicate as the source table for the second predicate.

Another source of unnecessary work is if the same expressions appear several places in the same query. To implement common sub–expression elimination we insert the sub elements of the query into a dictionary instance and duplicate entries would then refer to the same data. We could then restructure the query string such that the example in Listing 6.8 is transformed into the query in Listing 6.9.

```
SELECT ADD(t0.X,2), ADD(t0.Y,ADD(t0.X,2)), t0.Z
FROM vectors AS t0
```

Listing 6.8: A database query with common sub–expressions.

```
SELECT xAdd, ADD(t0.Y,xAdd), t0.Z
LET xAdd = ADD(t0.X,2)
FROM vectors AS t0
```

Listing 6.9: The query resulting from performing common sub–expression elimination.

The principle is basically to introduce the concept of variables into our query system. In LINQ, there is already a representation of this by using the 'let' keyword but ironically enough, the IQtoolkit takes the resulting expression an replaces all instances of the variable with the full expression. Thus, reversing this behavior would be the approach to implementing this.

While performance is the main goal of the optimizations described here, restructuring queries can also help reduce the problems with the lower floating point precision mentioned in Section 3.2. This is an issue which can be particularly problematic with scientific computing where precision is highly relevant.

The precision of a floating point operation can suffer particularly when subtracting operands with rounding errors such as the expression $x^2 - y^2$ (the inaccuracy is introduced by the multiplication). In such a case, the reduction can cause many of the significant digits to disappear, leaving only the inaccurate digits. Some expressions affected by this issue can be restructured to lessen or avoid the impact. With the above example, a better but equivalent representation of the expression would be $(x - y)(x + y)$ as this reduces the significance of the inaccuracies. For more information on this phenomenon, we refer the reader to the topic of *cancellation* in [37].

In addition to these concrete optimizations, any optimizations which applies for SQL might also apply for our DBMS. As this is a known problem which has been the topic of previous research, it should be possible to provide a solution which lends from this theory [38]. It might be possible to base an implementation on the query optimizers from existing DBMSs but it would be important to take into consideration the specific nature of the GPU as such optimizations are intimately tied to the executing platform.

In [38], they separate the types of optimizations into automatic optimizations such as query restructuring and manual optimizations based on suggestions for improvement. The latter should not be ignored as it is difficult to determine which optimizations to make, especially if the optimizer does not have all the information about the context of the query. Also, the optimizer should be able to take into consideration the context in which the database runs, as a system in development might have more lax requirements for computation overhead whereas a production database might not be able to spend as many cycles determining the best query plan. Query optimization is a complex topic and these issues just provide a few samples of what should be taken into consideration when implementing it for the DBMS.

6.6.3 Memory Allocation

As mentioned in [15], allocating large sections of memory can be costly and when performing operations on the database, large amounts of temporary memory are sometimes used. For example, performing a unary or binary operation will generally result in a temporary array of the same size as the source columns. Having multiple operations in an expression can cause this amount to increase even further. We do free up the used memory once an intermediary result is no longer needed, but we still pay the cost of the allocation at every step of the way.

One way we could reduce the amount of memory allocations would be to maintain a memory manager, much like the thread pool found in .NET. You would request an array of a given size from this pool and it would search for an existing array of the same size or larger. Performing several operations on the same table would result in temporary arrays of the same size so there should often be a suitable candidate. If the GPU is running out of memory, the memory manager can at any time free some of the unused data it holds on to. One consequence of this approach would be, though, that reused arrays contain old data and it would not be possible to assume an initial value of zero but this is more of a practical concern when writing kernels.

For tables which see a lot of modifications, the DBMS could benefit from a better approach to allocating memory rather than allocating a full new array on each new insertion. The test in Section 5.2 on appending data to tables verifies that this scheme introduces a significant performance penalty as inserting the same range of values becomes about 20% slower when moving from two insertions to ten.

We could use an allocation scheme of quadratic growth and shrinkage, such that when you need a bigger array you allocate double the memory and half the memory when shrinking beneath a certain threshold. This approach would support large amounts of small insertions or deletions to a table at the cost of some amount of wasted space, though at most half the array at any given time.

6.6.4 CUDA Code

Since CUDA version 2.1, it has been possible to perform just-in-time compilation of CUDA kernels using PTX code — a low–level assembler language for GPU kernels [39]. This capability makes it possible to tailor the kernels specifically to the given query, making it possible to merge kernels and avoid conditional statements in the code. For example, it would be possible to take the expression tree shown in Figure 6.1 and reduce it to a single kernel. The central part of the kernel would look similar to Listing 6.10.

```
int idx = //Current thread identifier
o_data[idx] = 2 * (i_data0[idx] + i_data1[idx]);
```

Listing 6.10: How several operations can be merged into a single kernel.

The main benefit from dynamically generating kernels in this way is that you have the result of intermediary operations directly where you need them avoiding the problem with expensive memory fetches mentioned in Section 3.2. Also, no more than a single temporary column has to be allocated as the intermediary results will be stored directly in the registers of the GPU rather than separate columns. It does not, however, allow us to dynamically allocate global memory inside a kernel so we still have to perform multi-pass operations for kernels with unknown result sizes.

To evaluate the potential of generating kernels at runtime we have manually implemented one which should yield a comparative performance. For this test, we execute a query with a number of math operations as normal as well the hand crafted equivalent, as shown in Listing 6.11. Figure 6.2 shows the execution times of the two queries with up to 4 million

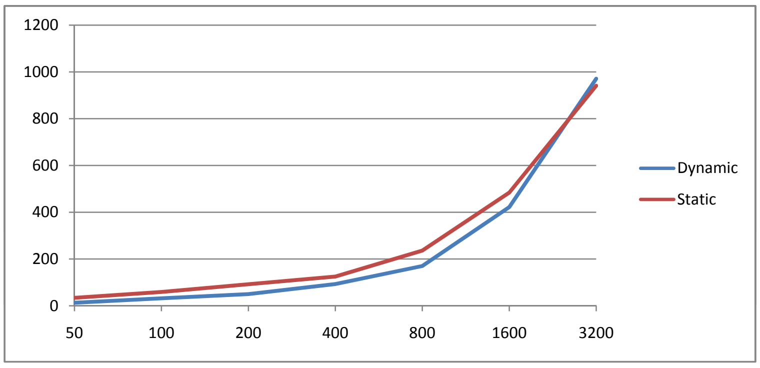

Figure 6.2: The execution time of the static and dynamic kernels.

vectors. Only the time spent on the math operations themselves has been included in the test. Note that the exponential growth is due to the quadratic scale of the x–axis.

```
var query = from v in vectorTable
            select Math.Sqrt(Math.Pow(v.X, 2) + Math.Pow(v.Y, 2) + Math.Pow(v.Z,
                2));
```

Listing 6.11: The LINQ query representing the database query executed as normal and as a single kernel.

The test yields several interesting observations. With a lower number of values, there is a clear performance increase when running a dynamically generated kernel. While it is not readily visible in the chart, the dynamic kernel starts out with a runtime of less than half that of the static kernels, a fact which stops being true at around 60,000 values. However, the gap closes the more values we operate with and at around 3 million values, the dynamic kernel becomes slower than the static ones. This did not match our expectations as we took the dynamic kernels for being faster in every regard.

We must highlight that this is just a preliminary survey of the potential for this technique and different math expressions could yield entirely different results. With that being said, what we believe has led to the unexpected results could be the order in which values are read and operations executed.

In the dynamic kernel, each thread would likely complete the entire math expression before another block gets to run on the same processor. However, in the static kernels, each operation is performed on all the values which would make the operation be cached in the registers along with the constant values we use. This, along with the fact that with enough values, the memory latency might be entirely masked could lead to a superior performance of the static kernels. The true potential of this technique is not clear from these test results, but we believe it shows significant promise.

Another aspect regarding optimization of kernels is the use of parallel computation primitives. As mentioned in Section 3.4.5, our CUDA code has employed a number of such primitives as building blocks to construct GPU algorithms. Some of the ones we found use for are *gather*, *scatter* and *prefix sum*. The more we use such primitives, the greater the benefit from optimizing them will be. Therefore, we suggest such optimizations to be of high priority as a bottleneck in a single primitive could have an impact on many algorithms. For reference, we mention in Section 3.4.2 how the *gather* and *scatter* operations can be optimized.

6.6.5 Indices

The current implementation of the index has a number of areas in which it can be improved. First and foremost, building the index on the GPU instead of the CPU would likely be faster, but more importantly, it would save the data transport to and from main memory. While the theoretical bandwidth of the PCI–Express bus is 4GB/sec, it is still far from as fast as on–device memory or even main memory. For example, a GeForce 8800 GTX GPU has a peak memory bandwidth of 86.4GB/sec [40]. As shown in the tests in Chapter 5.2, the overhead of creating an index is comparatively small, but on tables with more rows or with a larger type being indexed, this would grow.

The index is currently designed such that each thread block searches for one value. This was done as it is a fairly simple way of implementing the index which scales reasonably well. As the number of thread blocks per grid is 65536^2, this poses no practical limit to the number of values to search for. However, this design comes with some problems. As each thread block executes exactly once and independently from the other blocks, it is not possible to utilize shared memory which is faster to access by circa a factor 400 [1].

One way to use shared memory is to do searches in only one block, but store as much as possible of the tree from the root and down in shared memory. This would mean fast access to the nodes that are accessed the most. Currently, each thread block has access to 16KB of shared memory. With an index node size of 512, the first eight nodes would be stored in memory. While it does not seem like much, eight directory nodes is enough for over 2 million values to be indexed.

Chapter 7

Conclusion

In this project we have implemented a prototype GPU–based DBMS which supports some of the basic database operations. The DBMS is built using .NET and CUDA and can be accessed through any .NET language. Performing operations on the database is done using LINQ through a custom provider, turning queries into representations which the DBMS can understand.

In Section 2.2, we defined the following question as the basis for this project:

Is it possible to develop a DBMS residing in GPU memory using CUDA to facilitate a flexible data representation and computation model which can be accessed through a LINQ query interface, and if so, how does its performance characteristics compare to existing general purpose DBMSs for different types of tasks?

The most fundamental part of the question we have asked is "Is it possible to develop a DBMS residing in GPU memory using CUDA?". To illustrate that it is possible, we give and overview of the features supported by the current implementation of our DBMS – mainly through the LINQ provider:

- You can **create new tables** where each table stores objects of a specific type. When creating a table, columns are automatically made for each public writable property and only simple types are supported.
- You can **insert values** into a table.

- You can **select values** from a table to be moved to the host where the data is reconstructed as objects. Selections can use the standard math operators or math functions such as square root to create new results. The selected data can be mapped to arbitrary types as these are instantiated from the data, on the host.
- You can apply **aggregations** on columns of a table such as summation or average.
- You can **filter results** using comparison operators and boolean logic.
- You can **delete and empty tables** but not delete individual rows.

As evident from the above listing, we have not implemented joins but refer the reader to [9] for evidence that this works well on the GPU. We have only implemented a number of the functions on the `Math` class, but given that these were implemented (most of them would simply be wrappers over a similar CUDA function) the DBMS provides a very flexible and powerful computation model which can be applied using the same functions and operators developers are used to, only now integrated into a LINQ query.

Working only on members of simple types, we lack any sense of complex relationships but in Section 3.4.3 we provide a discussion of how such relationships could be implemented. Implementing nested complex types and join operations on different types of such objects, we would have a query model which covers the standard database operations.

Using parallel primitives, such as prefix sum, gather and scatter, as a core part of the DBMS has provided central places where optimization can be performed and affect many of the GPU operations. As the DBMS code base would grow, these common building blocks would also help make the code easier to understand and maintain.

7.1 Future Use and Adoption

As described in Chapter 6, there are many ways in which we can extend and improve the DBMS. However, one of the main strengths of the DBMS could very well be its ease of deployment and light–weight nature. While it could be beneficial to provide a more customizable DBMS, you can currently introduce an instance of the DBMS into your code in a single line of code – and the same applies for new tables. The reason it is so simple to setup your database is that the data objects you want to represent are understood natively by the DBMS and do not have to be translated into some other representation or, in other words, the data *is* the model.

Further supporting the role of our DBMS as a lightweight system is the low cost of inserting chunks of data shown in Chapter 5, which is a result of the lack of logging and ‘administrative’ tasks performed by ordinary DBMSs.

Does the DBMS look to be a viable solution when it has reached a more complete stage of development? To answer that question we look at the two main benefits of using our DBMS: computational power and ease of use. The tests performed in Chapter 5 show that, as expected, our DBMS provide superior computation capabilities compared to SQL Server. Combined with the light-weight insertions and deletions of data, our DBMS provides a good platform for performing computationally heavy tasks. Other tests, though, showed that we require optimization to be on par with traditional DBMSs, such as when selecting all values in a table. Unfortunately, we were unable to complete the filtered selection tests with an adequate amount of values, a severe problem, not only with regards to performance but also to be a functional DBMS.

For some environments, even if our DBMS would perform worse in some areas, the ease of deployment might prove the more important factor. To draw a parallel, dynamic languages have risen much in popularity in recent times, despite their comparatively poor performance. This is largely because of their ease of use and the numerous powerful frameworks which have appeared such as the popular Ruby on Rails.

While optimizations might change the picture somewhat, it is obvious that there are certain application areas where the DBMS is not an ideal choice. Small, incremental updates to the data stored in the DBMS are expensive which, as are selections in general. The index is unproved but does appear promising. However, as long as the focus remains on relatively expensive math on large sets of data without a need for high data integrity, it should be a good choice.

One concern for the future of the DBMS would be the reliance on CUDA which limits the use to NVIDIA GPUs. It is very likely that we would have to adopt a vendor neutral platform such as OpenCL[41], but for servers custom built for housing a GPU database this would not be a problem.

Chapter 8

Alternative Directions

All the work we have performed in this project has been directed toward developing a DBMS. However, is this the best view to take on the product we have implemented or could other abstractions benefit its future? The DBMS abstraction brings with it a lot of assumptions and expectations with regards to features and usage. They both force us, as the developers of the system, and the potential users of the system into a specific mind-set which could make it harder to identify some of the more innovative features this new technology could provide. If we abandoned the DBMS metaphor to use a more abstract concept for representing the GPU as a co–processor, people would, perhaps, be less inclined to stick to well known patterns and find novel uses and improvements.

On the other hand, there could be a danger with moving away from the well known in that many people might be hesitant to use it. Modern DBMSs are complex systems with numerous features, but is a comparatively well know set of features compared to those associated with GPGPU programming and have well established application areas.

In [7] we explored how to best make the GPU resources available to programmers in a simple and intuitive manner by overcoming the impedance mismatch between the two programming styles. One possible solution was to develop a DBMS as we have done in this project, but another was to focus on powerful data–parallel array types along the lines of those used in [15, 42], only with more extensive set operations. Without focusing on this directly, we have ended up with an internal representation of data–parallel arrays in the form of our table columns. Instead of going forward with the DBMS concept we could further explore this route.

We could, for example, develop a DBMS which also applied principles for working with data–parallel arrays such as map and filter functions, thereby creating a system which is immediately familiar to developers, but also facilitates new and useful use–patterns. It is not clear whether the grounding provided by such a well–known domain as databases hampers the evolution of the system, or if it benefits from a better understanding of its capabilities.

In the end, choosing one concept over the other might not have much influence on the implementation and features of the system unless some entirely new and better model were to be discovered. It would, however, shape the terminology we use for describing the product and how people think about it.

Given that we would continue developing the DBMS, one can also argue whether it makes sense to develop a DBMS rather than a query co–processor for an existing DBMS, the latter of which is a possibility discussed in [14]. In use cases like the one mentioned in Section 2.3, where data is always streamed to the GPU from, e.g., a DRDB, the co–processor model might be the better choice. However, for use cases where data is stored and worked on for longer periods, our DBMS would benefit from the lack of transportation between host and device.

The major benefit from developing a query co–processor would be the tight integration with the host DBMS, where it could supplement or add to its existing features. This integration comes at the cost of having to use it in the context of this particular DBMS, and this specialization makes the query processor less flexible. Developing a stand–alone DBMS allows freedom to provide a light–weight, flexible, and powerful way to perform computations. In principle, though, there is nothing preventing the co–development of two such products as they share many design and implementation principles.

Bibliography

[1] NVIDIA. CUDA 2.0 Programming Guide, 2008. `http://developer.download.nvidia.com/compute/cuda/2_0/docs/NVIDIA_CUDA_Programming_Guide_2.0.pdf`.

[2] Intel. Processors, 2008. `http://www.intel.com/support/processors/sb/cs-023143.htm`.

[3] Gpureview.com. ATI Radeon hd 4870 X2. `http://www.gpureview.com/Radeon-HD-4870-X2-card-572.html`.

[4] Larry Seiler et al. Larrabee: A many-core x86 architecture for visual computing. *ACM Transactions on Graphics*, 27(3), August 2008.

[5] Jon Stokes. Twilight of the gpu: an epic interview with tim sweeney, September 2008. `http://arstechnica.com/gaming/news/2008/09/gpu-sweeney-interview.ars`.

[6] Jingren Zhou and Kenneth A. Ross. Implementing database operations using simd instructions. *ACM SIGMOD*, June 2002.

[7] Morten Christiansen and Christian Eske Hansen. GPGPU Programming - GPGPU for the Masses. Technical report, Aalborg University, January 2009.

[8] J. Nickolls, I. Buck, M. Garland, , and K Skadron. Scalable parallel programming with CUDA. *ACM Queue 6*, 2008.

[9] Bingsheng He, Ke Yang, Rui Fang, Mian Lu, Naga K. Govindaraju, Qiong Luo, and Pedro V. Sander. Relational Joins on Graphics Processors. *SIGMOD*, 2008.

[10] Bingsheng He, Naga K. Govindaraju, Qiong Luo, and Burton Smith. Efficient Gather and Scatter Operations on Graphics Processors. *ACM*, (978-1-59593-764-3/07/0011), 2007.

[11] Naga K. Govindaraju, Brandon Lloyd, Wei Wang, Ming Lin, and Dinesh Manocha. Computation of Database Operations using Graphics Processors. *SIGMOD*, 2004.

[12] Christian S. Jensen, Hua Lu, and Bin Yang. Graph model base indoor tracking. *10th International Conference on Mobile Data Management*, 2009.

[13] René Hansen and Bent Thomsen. Using weighted graphs for computationally efficient wlan location determination. *The 4th Annual International Conference on Mobile and Ubiquitous Systems: Computing, Networking and Services*.

[14] Rui Fang, Bingsheng He, Mian Lu, Ke Yang, Naga K. Govindaraju, Qiong Luo, and Pedro V. Sander. GPUQP: Query co-processing using graphics processors. *SIGMOD*, pages 11–14, June 2007.

[15] David Tarditi, Sidd Puri, and Jose Oglesby. Accelerator: Using Data Parallelism to Program GPUs for General-Purpose Uses. *ASPLOS*, 2006.

[16] Ananth B. Brahma, 2008. `http://brahma.ananthonline.net/`.

[17] Stefanus Du Toit and Michael McCool. RapidMind: C++ meets multicore, 2007. `http://www.rapidmind.com/pdfs/DrDobbsJuly07-RapidMind.pdf`.

[18] NVIDIA. NVIDIA GeForce 8800 GPU architecture overview, 2006. `http://www.nvidia.com/attach/941771?type=support&primitive=0`.

[19] IEEE. IEEE standard for binary floating-point arithmetic, August 2008. `http://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=4610935&isnumber=4610934`.

[20] Intel. Intel and Floating Point. `http://www.intel.com/standards/floatingpoint.pdf`.

[21] TimesTen Team. In-Memory Data Management for Consumer Transactions, The TimesTen Approach. *SIGMOD 99*, 1999.

[22] Abraham Silberschatz, Henry F. Korth, and S. Sudarshan. *Database System Concepts*. McGraw-Hill, 5th edition, 2005.

[23] Kwang-Chul Jung and Kyu-Woong Lee. Design and Implementation of Storage Manager in Main Memory Database System ALTIBASE.

[24] Tobin J. Lehman and Michael J. Carey. A Study of Index Structures for Main Memory Database Management Systems. *Proceedings of the Twelfth International Conference on Very Large Data Bases*, 1986.

[25] Jun Rao and Kenneth A. Ross. Cache Conscious Indexing for Decision-Support in Main Memory. Technical report, Columbia University, December 1998.

[26] Jun Rao and Kenneth A. Ross. Making B^+–Trees Cache Conscious in Main Memory. *ACM*, (1-58113-218-2/00/051), 2000.

[27] Matt Warren. LINQ: Building an IQueryable provider series, April 2009. `http://blogs.msdn.com/mattwar/pages/linq-links.aspx`.

[28] Mark Harris. Optimizing CUDA, 2007. http://www.gpgpu.org/sc2007/SC07_CUDA_5_Optimization_Harris.pdf.

[29] FreeTDS - making the leap to SQL Server. http://www.freetds.org/.

[30] Bingsheng He, Wenbin Fang, Qiong Luo, Naga K. Govindaraju, and Tuyong Wang. Mars: A MapReduce Framework on Graphics Processors. *PACT 08*, 2008.

[31] MySQL AB. How MySQL Uses Indexes, May 2009. http://dev.mysql.com/doc/refman/5.0/en/mysql-indexes.html.

[32] Daniel Cederman and Philippas Tsigas. GPU Quicksort Library, November 2008. http://www.cs.chalmers.se/~dcs/gpuqsortdcs.html.

[33] Matt Pharr and Randima Fernando. *GPU Gems 2*. Addison-Wesley Professional, 2005.

[34] Jonathan G. Koomey. Estimating total power consumption by servers in the u.s. and the world. Technical report, Stanford University, February 2007.

[35] Gpureview.com. NVIDIA GeForce 8400M GT. http://www.gpureview.com/GeForce-GTX-295-card-603.html.

[36] Hiroyuki Takizawa, Katuto Sato, and Hiroaki Kobayashi. SPRAT: Runtime Processor Selection for Energy-aware Computing. *Proceedings of IEEE Cluster 2008*, 2008.

[37] David Goldberg. What every computer scientist should know about floating-point arithmetic. *Computing Surveys*, March 1991.

[38] Oracle. Optimizing the optimizer: Essential sql tuning tips and techniques. 2005. An Oracle White Paper.

[39] NVIDIA Corporation. Cuda directions, April 2009. http://www.cse.unsw.edu.au/~pls/cuda-workshop09/slides/05_CUDADirections.pdf.

[40] NVIDIA. GeForce 8800, 2008. http://www.nvidia.com/page/geforce_8800.html.

[41] Aaftab Munshi. OpenCL - Parallel computing on the GPU and CPU. *SIGGRAPH*, 2008.

[42] RapidMind Inc. RapidMind data sheet, 2007. http://www.rapidmind.net/pdfs/RapidmindDatasheet.pdf.

Appendix A

Summary

In this project we explore the possibilities for developing a DBMS which stores its data and performs calculations on it, on the GPU. The project has been started as a result of the high levels of computation power available on modern GPUs which is often used for little else than graphics and games.

To maintain a high level of performance we design the DBMS not to a general purpose system but rather one that is focused at certain types of jobs. GPUs excel at performing similar instructions on large amounts of data but as soon as you begin to perform lots of different types of operations and access your data in more random-access patterns, performance can drop dramatically.

To leverage the particular hardware architecture of the GPU we develop a DBMS which is best suited for working on large sets for generally static data. These operations could typically be complex data mining and pattern detection tasks. As such, the DBMS would often need to work in concert with a standard DBMS in which the data can incrementally be deposited and modified.

Using the CUDA framework from NVIDIA as well as C# we have designed and implemented an in-process DBMS which can be accessed from any .NET language. It is designed to make it as easy as possible to use, such that the computation capabilities become available to a broad range of programmers. The DBMS uses the .NET LINQ interface to construct queries and is thus easily to integrate into other .NET code, taking full advantage of type inference and Intellisense in Visual Studio.

The core of the implementation evolves around the representation of database columns which are essentially data-parallel arrays. When executing a database query, the DBMS then performs a number of parallel operations on this data to create temporary representations of the results. The execution of the operations happen on the GPU where the data is also stored. An overview of the supported functionality of the DBMS can be seen below:

- Creating new tables. This generates a column for each public, writable property on the type.
- Inserting values into tables.
- Selecting values from a table. Selections can use the standard math operators or math functions such as square root to create new results.
- Applying aggregations on columns of a table such as summation or average.
- Filtering results using comparison operators and boolean logic.
- Deleting and emptying tables but not deletion of individual rows.

To meet the goal of an easy-to-use database, you do not have to spend time crafting a database schema, instead the data *is* the model. This means that once you have defined you .NET classes to store in your database, your database schema is already defined.

What we have created should be seen as a prototype for demonstrating the viability of creating a GPU-resident DBMS. There are lots of features missing, but the most important for a sense of completeness are strings and database joins – though these have been proved to work well on the GPU by others. In addition to concrete features, the performance of the DBMS is varied as some parts have been implemented naively due to time constraints such as incremental insertions.

To aid with results filtering, we have implemented an index structure which is well suited for the GPU architecture. Cache sensitive search trees are stored in a single array and do not use pointers for navigation. Using an array structure makes it possible for the GPU to maintain a high level of memory bandwidth as it can fetch blocks of data at a time.

We have run a number of performance tests on the DBMS and compared the results to those of a Microsoft SQL Server 2008 database. The tests show that for insertions, our database is faster by roughly a factor 100, and deletions are performed in constant time. Selecting all elements in a table is 50%–60% slower than on the SQL Server database, but performing relatively complex math operations on the same data set is around 4 times faster using our DBMS. A problem caused by a naive implementation was verified to be a problem when appending values to an existing set of data, which suffered from an exponential growth in overhead.

The end product is a DBMS which is highly capable of analyzing large amounts of data but which has modest performance in the less ideal use cases. It is a prototype implementation and as such lacks certain important features, but with those implemented it is a flexible and light–weight, yet powerful solution.

Appendix B

Code Overview

In this appendix we provide an overview of the source code for the DBMS. All the code is contained in a Visual Studio 2008 solution containing the following three projects:

- DatabaseDemo
- GpgpuDatabase
- IQtoolkit

DatabaseDemo This project contains a single .cs file which performs a number of standard tests on the DBMS. This file serves as a good illustration for how the DBMS can be used in practice.

In addition, it contains all the CUDA code used by the DBMS and scripts for compiling them into .cubin files.

GpgpuDatabase This is the main DBMS source code which handles everything else than the LINQ query provider. The `GpgpuDatabase` class provides the way to interact with the DBMS.

The class `QueryManager` handles all the work with executing the database queries. It does not handle the concrete computations but it parses the incoming query string and evaluates it to provide the result of the query.

The `DataModel` namespace handles the representation of tables and columns and thus covers the storage aspects of the DBMS.

The `Linq` namespace provides all the custom versions of classes from the IQtoolkit which helps localizing the changes to the original framework.

Classes for representing query data and meta–data can be found in the `Query` namespace.

The execution of operations on the GPU is handled in the `Computation` namespace which provides wrappers over the CUDA.NET API, making GPU computations easy to invoke.

The `Index` namespace provides access to the implementation of the CSS tree index.

IQtoolkit This is the source code of the third part library which we have used as a basis for our LINQ query provider. Our project contains classes which inherit from key parts of the framework and most of the framework is unmodified.

The key modification to the toolkit is the class `MethodInternalizer` which captures method calls from the query, that would otherwise be executed on the host, and adds them to the internal database query representation, if supported by the DBMS.