Performance Evaluation of Query Processing Algorithms on GPGPUs

Vassil Hristov

Master of Science

School of Informatics

University of Edinburgh

2010

Abstract

Modern Graphical Processing Units (GPUs) can perform general purpose computing, next to standard graphical processing. Open frameworks, such as the OpenCL standard by the Khronos Group, enable developers to easily harness the computational power of GPUs. While in certain aspects, these are more powerful than standard CPUs, the latter are still a more suitable solution to many modern problems. In this dissertation we show the performance that can be achieved by evaluating database queries on GPUs using OpenCL. This is achieved by implementing a simple database model, capable of performing queries on any OpenCL-enabled device. The performance evaluation shows the strengths and shortcomings of this approach.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

(*Vassil Hristov*)

Contents

Listings

List of Tables

List of Figures

1 Introduction

Most modern computers come equipped with a multi-core CPU and trends show that future performance gains with respect to processing power will come from increasing the number of CPU cores, rather than increasing the clock speed of a single core. Furthermore, many computers also have dedicated graphics cards with multi-core GPUs and a high memory bandwidth for processing graphic data ([Kilgariff and Fernando, 2005]). Compared to CPUs, GPUs have significantly more cores, but these run at a lower frequency. Also, the architecture and instruction set are considerably different, compared to a multi-core CPU. Typically, GPUs are used to perform geometric operations and calculate the colour values of pixels on a screen. These tasks are run highly parallelised, since each pixel can be computed on its own. However, in recent days, graphics cards manufacturers have begun to provide APIs that enable developers of all programming areas to execute their own programs on a GPU. Thus, the processing power of graphics cards is no longer available only to developers with a good knowledge of the GPU programming model, but can be used for efficient processing of general computations, turning them into general purpose graphics processing units (GPGPU). Major software manufacturers have started incorporating these techniques in their software suites (e.g. Adobe Creative Suite 4 via plugins[1] or Cyberlink PowerDirector[2]). The interest in the power of graphics cards goes even further, including plans for building supercomputers based on GPGPUs. An example of this is the exascale computer, ordered by DARPA, which is expected to be ready in 2018 and is to exceed the performance of any current supercomputer[3].

While many have engaged in GPGPU-related research (see [Alcantara et al., 2009], [Satish et al., 2009], [Owens et al., 2007]), there are still many open issues regarding their applicability in different areas of computer science. In a database management system, there are many tasks that intuitively can be run in parallel to some extent, such

[1] http://www.elementaltechnologies.com/products/accelerator

[2] http://www.cyberlink.com/prog/company/press-news-content.do?pid=1828

[3] http://www.darpa.mil/news/2010/UHPCNewsRelease.pdf

as scanning tables, filtering and sorting data, performing certain join algorithms, as well as maintenance tasks, such as building indices or finding optimal query plans. However, for most of these tasks, computational power is rarely the limiting factor. For example, unless data is stored redundantly on separate hard drives, scanning the table from multiple threads will result in random I/O, which performs very poorly compared to a sequential table scan, and hence decreases performance rather than increasing it. Once the data is in main memory, further problems arise. Firstly, in many cases the bottleneck in performance will still be the time needed to transfer data from main memory to the processing units. Secondly, additional effort might be required to ensure no two threads are simultaneously modifying the same piece of data. The situation remains the same when using GPGPUs. There has been research done, investigating shared memory issues in parallel databases ([DeWitt et al., 1986], [DeWitt and Gray, 1992]), but the performance and applicability of GPGPU-based database management systems remain unaddressed.

Even though multi-core processors have been around for some time now, there are still few applications that make full use of their potential. The main reason for this is that algorithms have to be rewritten. In some cases this task is nontrivial or may even be impossible, since some problems remain inherently sequential. But since the increasing number of available processing cores is the main source for maximising performance, the potential speed-up of parallelised algorithms should not be neglected. Still, it is important to keep in mind that GPU cores should not be expected to perform as well as CPU cores on general tasks, the reason being the difference in their architecture. A modern CPU core will have a clock between 2 and 3 GHz, 2 to 4 MB cache, a complex instruction set and advanced features, such as out-of-order execution, branch prediction and more ([Ramanathan et al., 2006], [Guide, 2010], [Core 2 Duo, 2010]). In contrast, a GPU is clocked at around 1 GHz, has between 128 and 512MB VRAM memory and an instruction set optimised for floating point calculations and 3D rendering algorithms ([Kilgariff and Fernando, 2005]).

In this MSc project, we explore if and what performance gains can be achieved, by adapting query processing algorithms to make use of GPGPUs. We have implemented an example prototype system, identifying key difficulties and documenting their solution. This work extends the results from previous research to support and make use of the computational power of GPUs in the context of query evaluation. The main aim is

to examine the performance of the model system and compare it to the performance of a sequential query evaluating system.

The rest of this thesis is organised as follows. In the next section more information on some of the ideas and problems regarding parallel computing will be given. Section 3 describes our implementation of a database query engine that uses GPGPUs, as well as measures that have to be taken to prepare the execution of a query in this model. This is followed by details on query execution in section 4, and in section 5 we evaluate the work. Section 6 briefly points to related work in this area. The thesis is concluded by section 7, which provides a summary of the results and gives an outlook to future work.

2 Background

As already stated in the introduction, the key aspect of this thesis is running database queries in parallel on a GPU device. This section provides background knowledge for the techniques and technologies used during this project and it also gives some additional details, motivating the thesis.

2.1 Parallel Computing

Parallelism can often be found in modern software and hardware designs. In hardware, multiple units are used to perform tasks simultaneously, resulting in a faster or more reliable execution. There are many different examples, ranging from a lower level, such as multiple cores in a single CPU die, over a single system with multiple processors, up to entire systems, which are interconnected and can simultaneously work on the same task. The motivation for building parallel architectures results from the bounded performance of executing a single sequential task at a time. These bounds mainly result from limitations imposed by the laws of physics. As an example, in a single core CPU, the clock speed will mostly be bounded by the time, the signal needs to travel from one end of the die to the other, over wires and through logic gates (even if assuming a signal speed equivalent to the speed of light). Furthermore, the increasing problem of heat dissipation has to be solved for the ever smaller hardware units. Another issue with modern computer architectures based on the Von Neumann architecture is the bottleneck between memory and CPU, since both instructions and data have to be fetched in a certain order over the memory bus, which has a throughput that can hardly keep up with the frequency of a modern CPU. The parallel approach offers an alternative, bypassing some of the limitations of single processing unit architectures, offering a solution to the growing demand of more computational power and memory bandwidth.

To exploit the full potential of parallel architectures, the right software has to be used. In the past, as the computer technology evolved, systems did not support any parallelism and as a result, all problems were designed to be executed and solved sequentially. However, since hardware has been adapted to support parallel computations,

many problems have been remodelled to achieve higher performance on this type of hardware. But while optimised sequential algorithms exist for most problems, often their parallel counterparts have not yet been developed. One reason for this is the different types of parallelism, which all have different characteristics when it comes to achieving high performance. That is, to achieve high performance on a multiprocessor system, different techniques need to be used, as compared to using a chip-multiprocessor. Furthermore, splitting some problems into smaller elements that can be executed in parallel is not always a trivial problem and in certain cases, it is not possible at all.

Depending on the application scenario, there are four different types of parallel systems. Generally, a program consists of a set of instructions that are executed to alter some data. In a sequential program, one instruction is executed at a time, modifying one data item. Such setups are termed SISD for Single Instruction, Single Data and were the most common architecture up until the late 1990s. As mentioned before, most algorithms are designed to run on such a system. It is also an easy programming approach, since one can keep track of the system state at any point in time. In SIMD systems (Single Instruction, Multiple Data), the same set of instructions is executed simultaneously on different items of data. The most prominent example for such an architecture are graphics cards, which will process many pixels of the screen concurrently, while performing the same computation, but on different data. Consequently, a MISD system (Multiple Instructions, Single Data) simultaneously executes different instructions on the same piece of data, however, such architectures are rarely used in practice, since not many scenarios exist in which a MISD approach can provide any benefits. The final approach named MIMD for Multiple Instructions, Multiple Data, describes the architecture of a common chip-multiprocessor. That is, multiple instructions will process different data items at the same time. In the case of a chip-multiprocessor, each core will run independently from the other cores, processing its own set of instructions on its own data.

Parallel architectures are also distinguished by their memory model, which can be uniformly or non-uniformly distributed. In the first (uniform memory access or UMA), each processing unit is at the same distance from the main memory, as any other unit, whereas in a non-uniform memory access scenario (NUMA), each processing unit is close to a portion of the memory which it can access faster than the rest of the memory. In most cases, different threads or processes that run on different processing units and

alter different pieces of data, often need to exchange information. This can either happen by using different memory address spaces and messages for communication, or by operating in the same address space, directly altering shared data. The latter approach is, therefore, called a shared memory model.

Finally, parallel architectures can be categorised by the source of data and instructions that are used. In instruction-level parallelism, instructions from a single stream are executed in parallel. This type of parallelism is supported by superscalar processors and is managed by the hardware (the user has less influence on it) and is limited by dependencies between different instructions in the stream. With thread- or task-level parallelism, independent threads from the same application can run simultaneously, which means different instruction streams will operate on different data. To use this, the user needs to parallelise the algorithm and ensure correct synchronisation where needed. Finally, in data-level parallelism, the same stream of instructions simultaneously operates over multiple data streams. This is limited by memory bandwidth and is only applicable to certain algorithms with uniform data processing.

Parallel algorithms are generally more difficult to design than their sequential counterparts, as there are many pitfalls to avoid. As an example, running multiple threads simultaneously over the same data can result in unexpected behaviour. When all threads try to update the same object in memory at the same time (such as incrementing a counter), if no precautions are taken, following scenario might happen. Assume an MIMD architecture with two processing units and memory, shared between both units, where a value x needs to be incremented by two threads A and B, each running on one of the processing units. The following is an example sequence of the operations.

```
t0: A read x
t1: B read x
t2: A x = x + 1
t3: B x = x + 1
t4: B write x
t5: A write x
```

There are several important issues to consider when looking at the above example, and they depend on exactly how the operations will be executed by the hardware. The first thing to consider is reading the value x. While it is said that both threads run simultaneously, it is likely that the hardware will not allow both processes to read

the desired memory address at the same time, but will synchronise the access. The consequence of this is that one thread will always be served before the other, yet it is not clear which one will actually be the first one to read and which will come second. Hence, it is possible that each time the code is executed, a different thread will be served first. The same applies for writing the value back into memory. When both processes have finished and written data back into memory, the value for x will be *x+1*, even though the correct result should be *x+2*, as two increments take place. To avoid this problem, explicit actions need to be taken by the program to ensure the correct result. One of the options is to use the mutual exclusion (mutex) construct, given that the hardware supports it, which will ensure that only one thread will execute a certain set of instructions covered by the mutex. For the given example, all three instructions of each process will have to be in mutual exclusion. Another common feature supported by the hardware is called atomic memory operations. This is usually a special instruction set that guarantees that a certain operation, will be executed as a single instruction. Such an instruction could be `increment(x)`. Both processes would then make a call to this function and can safely assume that the correct value of x will be stored in memory after its execution.

While with both options one can achieve correct results, they also imply a point of serialisation, where one process will have to wait for another process to finish. Such serialisations may have a major impact on the performance of a parallel program and in some extreme cases, the performance gain of parallelising the problem might be completely lost. Hence, avoiding them is an important aspect when designing parallel algorithms. For example, it might be better to have some additional overhead in each of the parallel threads (for example additional bookkeeping), if that helps avoid synchronisation. Another issue that might arise in shared memory systems is cause by caching, where several caches might hold the same data, however, most modern hardware implements cache coherence protocols that will ensure a correct system behaviour.

2.2 GPGPU vs. CPU

To run parallel algorithms, special hardware that supports parallel computation is needed. Two of the most common examples are modern chip-multiprocessors, such as the Intel(R) Core(TM) 2 Duo by [Intel, 2010] with two chips on die, or the AMD

Opteron 6100 Series for servers with up to twelve cores by [AMD, 2010]. The other even more common example of hardware supporting parallel computing are graphical processing units, such as the NVIDIA GeForce 9600 GT by [NVIDIA Corporation, 2010] or the ATI Radeon HD 4600 Series by [AMD, 2010], with up to 320 processing units.

Until recently, GPUs and CPUs were dedicated to different disjoint tasks. CPUs were the main processing power in a computer system, responsible for the execution of all programs, while GPUs only emerged as supporting devices, responsible for preparing graphical output data. However, due to the quick development of GPUs, an interest in harnessing this power appeared and is growing even more, now that more accessible programming interfaces have emerged (see next section). If we were to compare both architectures based on clock frequency, number of computation cores and memory bandwidth, an off-the-shelf GPU will easily outperform even the fastest CPU available ([Owens et al., 2007]). This difference in performance is a result of the different focus of the two architectures. CPU manufacturers have optimised their chips to run sequential code, using multiple cores mainly for instruction-level parallelism, but still supporting task- and data-level parallelism. Meanwhile, GPU manufacturers have developed chips to efficiently run data parallel programs, but with a poor support for sequential programs and limited to no support of certain features, available in every CPU. These differences, together with some other distinct architectural features, make it hard to accurately guess which approach is better suited for certain application.

The following are some of these key differences:

- **Memory bandwidth**: While a GPU chip may have a very high memory bandwidth (57.6 GB/sec for the GeForce 9600 GT, as given by [GeForce9600GT, 2008]), this only concerns shipping data from the video memory to the graphics chip and back. To read data from main memory, the bandwidth is limited by the speed of the bus, to which the card is connected. For a PCI Express bus of lane width 16, this limit is 8 GB/sec and is shared by all devices on this bus. Depending on clock speed and RAM type, CPUs will mostly have a similar bandwidth.

- **Cache**: Modern CPUs are usually equipped with two levels of cache, which greatly increase the perceived bandwidth and latency, where 30GB/sec transfer rates and more can be expected (see [Zack Smith, 2010]). GPUs, however, do not have caches, but only limited amounts of shared memory that is in fact faster than the

global video memory. However, this does not operate as a cache and data has to be copied to and from this memory location explicitly by the program.

- **Context switching**: On a CPU, switching contexts between different threads is done by the operating system and is a relatively slow process that requires several clock cycles. GPUs, however, implement the context switching in hardware, which greatly reduces the time it takes to switch from one thread to another on a single processing unit.

- **Advanced computational features**: As already mentioned, CPUs support a wide range of advanced features, such as branch prediction, out-of-order execution, pipelining, etc. All of these features do not make much sense in the context of graphical processing and are accordingly not supported by GPUs.

Due to these differences, it is necessary to conduct thorough experiments to decide whether or not any application will have a significant increase in performance when using a GPU instead of a CPU for the main computational tasks. [Lee et al., 2010] provides an analysis of the performance of some common applications, when run on both a CPU and a GPU. In this thesis, the performance of a model database is examined to help decide, if GPUs can be used to increase the performance query evaluation.

2.3 OpenCL

The Open Computing Language (OpenCL) is a standard maintained by the Khronos Group ([Khoronos Group, 2010]) for running parallel code on CPUs, GPUs and other devices capable of performing computations. It provides a programming language, which is an extension to the standard C language (C99 specification) and is used to define both data-parallel and task-parallel algorithms. However, the language has both some extensions and some limitations, compared to the C standard. Additionally, the framework provides an API for preparing, coordinating and executing parallel programs.

Initially OpenCL was developed by Apple Inc. ([Apple Inc., 2010]), but the final proposal of OpenCL 1.0 was designed by the Khronos Group, whose members include most of the major chip manufacturer and software developer companies (Apple, Intel, NVIDIA, IBM, AMD, Samsung, Nokia, and Adobe to name a few). Version 1.1 of OpenCL was only released after the work on this dissertation had started, hence ver-

sion 1.0 is used. However, a majority of the changes proposed in 1.1 are not relevant for this thesis.

The most important aspect of OpenCL is that a programmer writing a parallel program, can be unaware about what device their program will run on, as long as it supports the OpenCL standard. Other languages that provide access to a GPU for instance, require developers to have a certain knowledge about the execution model, memory addressing and other aspects of the graphics card that will be used. Not only is this not a requirement for writing OpenCL code, but it is also possible to run the same code on a CPU, GPU, or on embedded chips simultaneously from within a single application. To support this behaviour, the code running on a device is compiled just-in-time during runtime for the specific device it will be executed on, to ensure the same version of software will run on all enabled devices.

2.3.1 Platform Model

First we discuss the OpenCL platform model, which will then be used to provide a detailed description of the execution model.

Host The host is the logical unit that will perform OpenCL calls. The host device (commonly the CPU, processing instructions from main memory) will run a host application, in which all relevant data and objects for issuing calls to an OpenCL device, are prepared and executed. The host is responsible for the flow of the entire application.

Devices and Compute Units According to the OpenCL specification, in a computer system, any hardware that can perform any computations is called a device. Each device has one or more compute units. Each compute unit is capable of interpreting instructions from an instruction set, independently from other compute units on the same device. As an example, a dual-core CPU would be considered a device with two compute units in terms of the OpenCL specification.

Context The context is defined and managed by the host application. It is used to manage all relevant data, such as devices, programs, kernels, command queues and memory objects. OpenCL programs run within a defined context and may access rele-

vant data and resources.

Program A program is used to define kernels and functions that can be called by the kernels defined in the same program, as well as type definitions (custom structures, etc.), used by the former. Each program is built for a specific context, on which its kernels will be executed. It is possible to build any number of programs for a context. The source code is written in the OpenCL-C language. While it is possible to build multiple programs in one context, kernels and functions from foreign programs are not accessible. That is, if program A defines a function X and program B defines a kernel Z, it is not possible to make a call X from within Z. Hence, if a kernel will need to execute a function, both have to be defined within the same program.

Kernel A kernel is a function that can be executed on any OpenCL compliant device. It resembles a standard C-function and performs any sort of computation with the provided arguments. These can be either simple types, or pointers to a memory location on the device. All kernels have the return type `void`, meaning that all results have to be written back into memory. All kernels are compiled during runtime, from the source found in a program, already built by the host. Kernels are sent to a device and assigned to compute units, which will execute it in parallel, independently from other units.

Work item The work performed by a single compute unit is called a work item or thread. Each work item will execute a certain kernel: in a data-parallel scenario, all work items will be running the same kernel, and in a task-parallel scenario, different work items will be running different kernels. While it is said that all threads run in parallel, in most cases the number of threads that are required for a certain problem will be higher than the number of compute units available on a device. In such a situation, a device with n compute units, will execute n threads at a time. It is important to know that when a group of threads is concurrently executing the same kernel, every work item will either execute the same instruction as all other work items, or it will be idle. That is, if a kernel contains conditional branching, such as an if-statement and only a subset of the kernels enters this statement, all other threads will remain idle, until the if-statement has been processed, after which all threads can resume executing instructions simultaneously.

Work groups Work groups are sets of threads which run simultaneously on one device and execute the same kernel. There are two types of groups: global and local. The size of the first corresponds to the total amount of work items that will be started in order to process the kernel, whereas the latter is a smaller group of threads, where the size of the global group is a multiple of the size of the local group. The important difference between the two groups is the ability to synchronise different threads by using memory fences or barriers. In case of a global work group of size n and local group of a single thread, firstly there are no guarantees on which thread will execute when, that is $x_n < x_1 < x_0 < x_{n-1}$ and $x_0 < x_1 < x_n < x_{n-1}$, where $a < b$ indicates that thread a is executed before thread b, are both possible execution orders. Secondly, barriers and memory fences cannot be used to synchronise threads. In a local work group, there is still no guarantee about the execution order of threads within the group, however, only threads from one local group will run concurrently and finish, before threads from another thread group are scheduled. Furthermore, barriers and memory fences can be used to ensure all threads have reached the same state. For most devices, the size of the global work group will be 2^{32}, the size of the local groups, however, will be by far smaller (4 to 512 threads, depending on the device). The standard also supports two- and three-dimensional work groups. As an example, a two-dimensional group of 32 work items could simultaneously perform an operation over a 32x32 matrix, where each thread would process one element of the matrix.

Memory object As already mentioned in the kernel section, memory objects are used to provide input data to kernels and read the output data back to the host, that is they can be used to exchange data between the host and the devices. Two different types of memory objects are supported in the current OpenCL standard: buffers and images. Buffers are used to store any kind of data (such as integer or float arrays, text, or custom structures), and images, which, as the name suggests, are optimised for storing two or three dimensional image data. Since the latter are not relevant for this thesis, further details are omitted. Memory objects are initialised on the host and later sent to the device, since kernels can only access the memory of the device they are running on. More information on memory management is provided in Section 3.2.

Command Queue Once a kernel is built and its arguments are set, it is put into a command queue, which will ensure its execution on the related device. Kernels are executed in the same order they are put into the queue. However, queues have two modes of execution – in-order and out-of-order command execution. In the first, each command is completed before the next one will start. In the latter, commands will start in the order they were added to the queue, but a command can also start executing before its predecessor has finished. The API provides functionality, to ensure synchronisation when the queue is operating in out-of-order mode, when it is needed. Finally, the queue can also be used to schedule memory operations, such as copying data to or from the device.

2.3.2 Execution Model

The above described notions are the basic objects used to run a OpenCL application. Following, the description of the execution model will provide details on how these objects are set up and how a program is executed on a device. Listing 1 shows a simplified outline of a program, running a kernel on an OpenCL-enabled device.

```
cl = initOpenCL();                                    // Setup device, context and queue
comp = loadKernel(cl, "program.cl", "comp");          // Load and build a kernel
mem_input = createBuffer(cl.ctx, r);                  // Prepare the memory object(s)
mem_output = createBuffer(cl.ctx, w);
setKernelArguments(comp, mem_input, mem_output);      // Set the kernel arguments
enqueueKernel(cl.queue, comp, 128, 16);               // Trigger kernel execution
res = readBuffer(mem_outpu);                          // Read the result
```

Listing 1: Running an OpenCL program

In the first lines, the OpenCL context is initialised. During this phase, a suitable device is found on which the application will run. The programmer can specify the type of device that is preferred, or even load and use multiple devices. For the given device, a context and a queue are set up. In the second step, a program is loaded and built, which contains the definition of the kernel `comp`. The latter is compiled for the device, specified in the context. As a next step, the memory objects are prepared – one for the input and one for the output, the first being marked as read-only, since data from the input will only be read and no in-place modifications will occur, and the second consequently being marked as write-only, since data will only be written to this memory object.

Further options for managing memory are available and will be discussed in more detail in Section 3.2. After setting the memory objects as arguments of the kernel, the actual execution is triggered. A total of 128 threads will be invoked, running in groups of 16. After the computation has completed, the result is copied from the device memory to host memory. For a real code example, please refer to Appendix B. Further information on OpenCL is provided in [OpenCL 1.0 Specification, 2009].

CUDA As an alternative to OpenCL, NVIDIA graphics cards support the CUDA (Computer Unified Device Architecture) framework, which offers very similar functionalities. It brings the advantage that it supports both C and C++ and hence, it can be easily integrated into existing projects. However, a major shortcoming is the fact that it is limited only to graphics cards manufactured by NVIDIA, and also it is not an open standard, but is entirely maintained by NVIDIA. Since OpenCL supports any kind or make of processing units and it is an open standard, this thesis uses this framework, rather than CUDA. For more information on the latter, please refer to [CUDA, 2010].

3 A Database Model Using OpenCL

In this section, we describe the database query engine model, which is used to execute queries over given data, using parallel algorithms and OpenCL. Also, general information on memory management and input preparation is given later in this section.

3.1 Database Model

Since the OpenCL framework offers C bindings, we implemented the model system in C as well. An important advantage of using C over any other programming language of higher level, is the low level access to memory management that is useful when preparing input and output for each query. There are two aspects of the model, which are discussed hereafter.

The first aspect is the database relevant objects. There are three key structures, used to handle data in the model system: tables, tuples and attributes. Each of these structures comes in several flavours. The key object is a tuple, which consists of several attributes and can be grouped with other tuples to form a table. The model supports four types of attributes: integers, long integers, floats, and strings of 8 characters length. Initially, a double type was intended, yet in the OpenCL standard, double values are optional. While on the machine, on which the model was developed, the graphics card supports double values, their size on the device differed from size on the host: on the latter, a float value is 4 bytes long and a double value is 8 bytes long. On the device, however, both types are sized 4 bytes. Tuples are stored in an array and accessed via a table object that holds a reference to this array. Furthermore, each table object holds the total number of tuples it refers to. Finally, the model supports a predefined table structure. The schema that is used is based on the Wisconsin Benchmark as introduced in [DeWitt, 1993], with a small modification. Appendix A shows the definition of the table. In the original benchmark, there are only integer and char fields, however, in this modified version, we defined the four fields `onepercent`, `tenpercent`, `twentypercent` and `fiftypercent` as floats, rather than as integers. Upon execution, queries can access a table and its tuples to compute the output. While support for different table schemas is feasible, this model uses a static schema for a better performance.

The second aspect of the model system is the interface for accessing OpenCL features. For convenience, a custom structure is created that holds the relevant objects, such as device ID, context and command queue. These are initialised upon the start-up of the system and destroyed when the system is shutting down. This means that the OpenCL context is initialised only once and then can be used repeatedly in different queries. Since this context object also holds all programs that are used in any of the available queries, programs also do not need to be built on demand by each query. However, in a real life system, this approach would be only limitedly possible, since, as can be seen later, most queries will require a very specific kernel. To support this in a real life scenario, kernels need to be generated on the fly for every query, yet this is not expected to result in a high degradation of the performance, since the just-in-time compiler, used by the OpenCL framework, will compile most kernels quickly enough. Furthermore, once a kernel is generated, its binary representation can be kept in memory for future use. Building the required kernels could be based on the optimal plan an optimiser has picked. To strike a balance, in the implemented model we only load and build programs on startup, but generate the kernel objects on demand. The OpenCL context uses only one device, which can be specified during start-up. Instructions for this device, are sent over a single queue, which executes them in-order. Finally, some convenience functions are available, which use functions provided by the OpenCL framework and the implemented OpenCL context object, providing easier access to frequently used functionality, such as compiling kernels and freeing memory.

As already mentioned, the system supports a certain set of queries. All queries are predefined and static, meaning they will always perform the same operation, when provided with the same output. For example, a query performing a selection will always use the same predicates. As a result, when provided with a specific input, all queries will always deliver the same results. Depending on its complexity, a query will consist of one or more kernels, which, executed in the right order, will process the input and generate the correct output. While supporting some dynamics in the queries, such as using different predicates in a selection, or joining two tables on different attributes, would be a straightforward matter, it has been omitted in this thesis, as it is not expected to provide additional relevant information on the quality of the model under consideration. For some of the queries, a brief description will be provided on what changes are due, to generalise them.

3.2 Memory Management and Input Preparation

Section 2.3.1 introduced the basic memory architecture of OpenCL. Following, some more details are provide on how and where memory is allocated for queries. As mentioned previously, kernels can only access data that is present in the device memory. Each OpenCL device supports three types of memory: global or constant, local, and private. Each object, referenced in a kernel, will reside in one of these memory areas, which will be denoted by the corresponding address space qualifier. The global memory is the largest, but slowest memory area and it is available to all work items in the global work group. A modern GPU will usually have between 128MB and 512MB of global memory. Variables declared as `__constant` are also located in the global memory pool, but can only be read from, that is, writes are not permitted. The local memory is both smaller (16kB for modern GPUs) and faster than the global memory and is accessible to all elements in a local work group. This rather small amount of memory may impose a further limit on the size of local work groups. This can be the case when all elements in the local group need to write a certain amount of data to local memory. For example, if all work items need to allocate 8 `int4` variables in local memory, the maximum number of threads in the local group will be bound to 128 (since $128 * 8 * 4 * 4 = 16384$). Finally, private objects are only accessible to a single work item and are also the fastest and smallest memory location. This is the default address space for objects created within a kernel.

Moving data between different memory address spaces is done explicitly by the programmer. Therefore, to provide access for a kernel to a tuple, the latter must be copied explicitly from the host memory to the global memory by the host application, and in some cases, copied from global memory to local memory by a kernel. There are several options for providing input to a kernel, the basic idea being copying data from the host to device memory. The easiest way to this is to allocate the same amount of data on the device that the input requires and then instruct OpenCL to copy it. However, this approach has the following drawback: upon scheduling the command, all data is shipped to the device, regardless what portion of it will be used. Common database queries require only a few attributes to compute the result, while the remaining attributes are only copied to the output. Consequently, given a query that computes the result based on a single attribute, the remaining data is copied to and from the

device, without being used at all. An alternative solution to this is to leave the input on the host and let OpenCL copy only the required data on demand. In this case, the global memory is used as a form of cache, where relevant portions of the input are copied from and back into host memory. With this approach, still the same amount of memory needs to be allocated on the device, as will be read from the host memory. Clearly, the memory a GPU offers will not be enough for processing large data sets in one sweep, which is why in such scenarios, the input needs to be partitioned first and then, when each partition fits into the device memory, it will be processed separately. The final result can be computed by merging all partial results. For the later described algorithms, different techniques have to be employed to achieve the correct results, which will be discussed for each algorithm. For a complete list of memory allocation options, please refer to [OpenCL 1.0 Specification, 2009].

Another related problem that needs to be considered is the lack of dynamic memory allocation on a device, which is especially relevant for writing the results of a query. This means that all memory has to be pre-allocated before a kernel is executed. While for some queries, the size of the output can be computed prior to its execution, for other queries this is not the case. In the query descriptions to follow, we address this problem and describe methods, used to overcome this problem.

4 Query Evaluation with OpenCL

After all the required background knowledge has been provided in the previous sections, following several different algorithms will be described that are common to database management systems. Each of the algorithms can take advantage of data parallelism in a different manner and for some, additional effort is required, compared to the sequential versions of the same algorithm. However, all queries employ data-level parallelism, as it is the nature of a database management system to perform certain operations on a set of tuples. Furthermore, these are only examples that should cover the most common tasks in a database system. While there are numerous other operations that are not explicitly mentioned here, all of them can be implemented in this model and hence it can be extended to support the full feature set of a real life database system.

4.1 Projection

In a database system, when performing a projection π over a table T with a schema $\mathcal{S}$, the output will be a new table T' with a schema $\mathcal{S}'$, where $\mathcal{S}' \subset \mathcal{S}$. For example the result of π_{AC} with $\mathcal{S} = \{A, B, C, D\}$ would be $\mathcal{S}' = \{A, C\}$. A projection does not have an impact on the cardinality of the output. The algorithm we used in this evaluation is slightly more complicated, as it takes two attributes from the source schema, computes their sum and returns it as a new attribute. The equivalent SQL statement can be seen in Listing 2. Both attributes, referenced in the query, have unique numerical values.

```
SELECT t.unique1 + t.unique2 AS summation
FROM some_table AS t
```

Listing 2: Project query

While this is a really simple scenario, it is a perfect example of data-parallel programming. Each tuple in the input can be processed independently from all the other tuples. Hence all that a kernel needs to do is read a single tuple, compute the sum of both attributes and write it to the output. To execute the kernel, two buffer objects need to be created – one for the input and one for the output. The buffer for the input will have the size of the table that will be processed and will actually use a pointer to

the same memory location on the host. The output buffer will also use host memory to store the results, hence the required memory needs to be allocated beforehand. Since the input buffer will only be read and the output buffer will only be written to, we use corresponding flags when creating the buffers, to let OpenCL know that one buffer will be read-only and one will be write-only. Respectively, given the device supports the `__constant` memory location modifier, the input can be defined as such. The entire code for this example can be seen in Appendix B. Once memory objects and kernel are set up, the execution can be started, using an automatic value for the local work group size (if no value is provided, OpenCL will try and guess the best value), and the global work group size equal to the number of tuples in the input. That is, one work item (or thread) will be started for each tuple in the input and hence the global thread ID can be used as the offset of the tuple in the input that will be processed by this thread.

```
__kernel void
sum(__global db_tuple *input,
    __global db_attribute_numeric *result)
{
        int gid = get_global_id(0);
        result[gid] = input[gid].attrb_num[0] + input[gid].attrb_num[1];
}
```

Listing 3: Project kernel

As already said, this is a very easy example, however it uses the full potential of parallel computing, since no synchronisation is required at any stage. Each work item can be executed at a random point in time, without interfering with the other work items. Memory conflicts cannot occur, since every thread is reading from a different location in memory and writing to a different location in memory. The only problems that can occur in this version of the algorithm are memory bank conflicts. These can occur when two or more threads try to access two distinct elements that both are allocated in the same memory bank, which causes performance degradation, since access to single banks is serialised. Using special addressing techniques, this issue can be avoided.

To enhance the algorithm to support arbitrary projections, two changes need to be done. Firstly, the memory, allocated for the output, needs to match the size of the result, that is it will be the product of the number of attributes, their respective sizes

and the total number of tuples in the result. Secondly, the kernel has to be adapted to copy all relevant data in its designated place in the output buffer.

Finally, the given example only supports input that fits entirely in the maximum allocation size of a device. Clearly, in certain cases this will not be enough. Hence another modification of the algorithm is due. Before its execution, the size of the input has to be inspected and if its found to be larger than the maximum, it has to be broken up into partitions, where each one of them is smaller than the maximum allocation size of the device. For the projection this is a rather simple process, since there are no dependencies between different partitions. As a result, the host application will loop over all partitions, which map different offsets of the input, and in each iteration, a portion of the output will be computed.

4.2 Scan

The next algorithm that we discuss is generally not related to a specific query, but is used by some of the remaining query algorithms and hence will be described here. The prefix sum or scan algorithm operates over an array of elements, building the partial sum for each element. That is, after the algorithm completes, an element x_n will contain the value $\sum_{i=0}^{n} i$ for an inclusive scan or $\sum_{i=0}^{n-1} i$ for an exclusive scan. While intuitively, building the sum of consecutive elements seems to be an inherently sequential task, it is possible to perform the algorithm in parallel. The prefix scan can easily be extended to support any associative operation, as we will see later.

Generally, the algorithm will run in two phases, the first being called up-sweep or reduction and the second down-sweep. When looking at Figures 1 and 2, the elements that are modified in each step, when looked at together, resemble a binary tree, which is traversed from the leaves to the root in the first phase and from the root to the leaves in the second phase, hence the naming of the two phases. As already mentioned, the algorithm comes in two flavours: inclusive and exclusive, depending on wether the total sum will include the value of the last element or not. The first phase for both versions is the same, however, the second phases are different. Listing 4 shows the pseudocode for the two versions of the scan. The idea behind this algorithm is to build partial sums in the first phase and use these to compute the value of every element in the second phase. Apple provides a sample implementation of an exclusive scan in OpenCL

in [Apple, 2009]. In [Harris, 2007] an implementation in CUDA is given by NVIDIA, which also includes a very detailed description of the code.

```
1  /* Reduction */
2  for d := 0 to log2n − 1 do
3      for i := 0 to n/2 do in parallel
4          if i ≥ 2^d − 1
5              x[2^d * (2i + 2) − 1] := x[2^d * (2i + 1) − 1] + x[2^d * (2i + 2) − 1]
6
7  /* Down−sweep inclusive */
8  stride := n/2
9  for d := 1 to log2n − 1 do
10     stride := stride/2
11     for i := 0 to n/2 do in parallel
12         if i < 2^d − 1
13             x[stride * (2i + 3) − 1] := x[stride * (2i + 2) − 1] + x[stride * (2i + 3) − 1]
14
15 /* Down−sweep exclusive */
16 x[n − 1] := 0
17 for d := log2n − 1 to 0 do
18     for i := 0 to n/2 do in parallel
19         if i > 2^d − 1
20             temp := x[2^d * (2i + 1) − 1]
21             x[2^d * (2i + 1) − 1] := x[2^d * (2i + 2) − 1]
22             x[2^d * (2i + 2) − 1] := x[2^d * (2i + 2) − 1] + temp
```

Listing 4: Reduction phase of the scan algorithm

The entire scan algorithm operates in local memory. For that, the relevant data is copied from global to local memory before the reduction phase, and after the respective down-sweep phase, the data is copied from local to global memory.

Reduction To build the partial sums of an array of size n, we require $n/2$ work items. Each one of them goes through a loop (line 2), where in each step the sum of two elements is computed. In each of the following iterations, only half of the previously running threads is running, while the others remain idle, until in the last step, only one thread is executed (line 4). In the first step, the sum of two neighbouring elements is computed, in the second step, elements with a distance of two are added, and so on, until in the final step the sum of the middle and the last elements is computed. The

algorithm operates in-place, that is the new results are stored in the same array. Figure 1 depicts this phase of the algorithm when processing an array of size 16. Each iteration of the loop is marked with the value of d on the left. A group of two arrows represents the read-write operations of one work item. The darker the area of an element is, the more elements it has accumulated. For example, in the first step ($d = 0$), thread 0 reads elements 0 and 1, computes the sum, and stores it in element 1. So do threads 1 to 7, operating on the respective elements. In the next step, $d = 1$, thread 0 reads elements 1 and 3 and writes the sum in element 3. Threads 1 to 3 proceed as well, however, threads 4 to 7 remain idle. After another two iterations, only thread 0 is running, storing the sum of elements 7 and 15 into element 15.

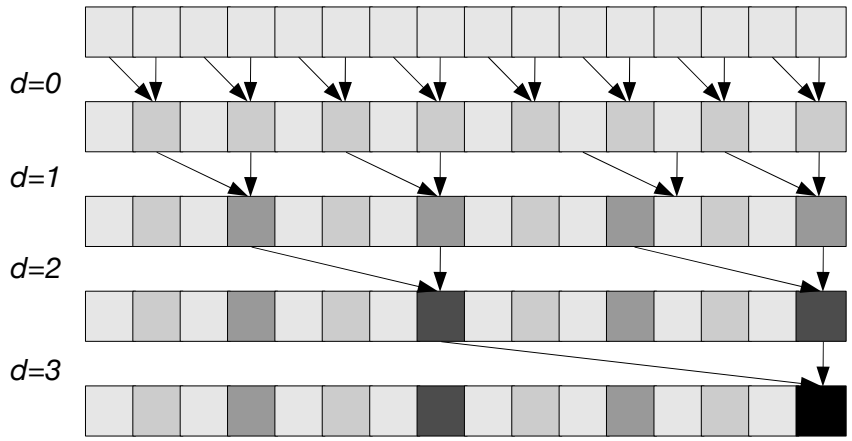


Figure 1: Reduce phase of the scan algorithm

After this phase of the algorithm is completed, the last element of the array contains the total sum of all of its elements. Furthermore, every second element holds an intermediate sum, and more precisely every $2n * 2^m$-th element with $n, m \in \mathbb{N}$ contains the sum of the preceding 2^m elements. The result of this phase is used as input of the two variations of the down-sweep phase and for an array of size n it is computed in $log_2 n$ steps by $n/2$ threads. The model also supports a modified version of the algorithm that can operate over a certain element of an integer vector array, which as can be seen later, is useful in certain scenarios. This modification, however, only concerns the data that is copied from global to local memory, creating a single integer array as input for the reduction.

Inclusive down-sweep In the final result of the inclusive down-sweep phase, each element of the array contains the sum of the value it had initially and all the values of its predecessors. The algorithm basically walks a similar path as in the up-sweep phase, but instead of computing an intermediate result for elements, it uses the already computed partial sums to compute the final values of some of the elements. These can then be used to compute the final values for further elements, until in the final iteration, half of the elements already contain their correct values, which are used to compute the correct values of the remaining half. In the reduction phase, each step would half the amount of threads that can run simultaneously, whereas in this phase, each step approximately doubles the amount of threads that can be executed concurrently. Figure 2 illustrates how this phase would execute, provided with the output that was generated by the reduce phase. Again the darker an area is, the more elements it has accumulated. Before the execution starts, there are already two fields with correct values: the middle and the last field. Since the final value of a field is only relevant for succeeding elements (that is elements to the right in the figure), the last field is of no interest to us, since it does not have succeeding elements. As in the reduction phase, each work item is illustrated by two arrows, which represent reading two memory locations and storing their sum in one of the locations. Consequently this phase also runs in place and processes the array in $log_2 n - 1$ steps with a maximum of $n/2$ concurrent threads.

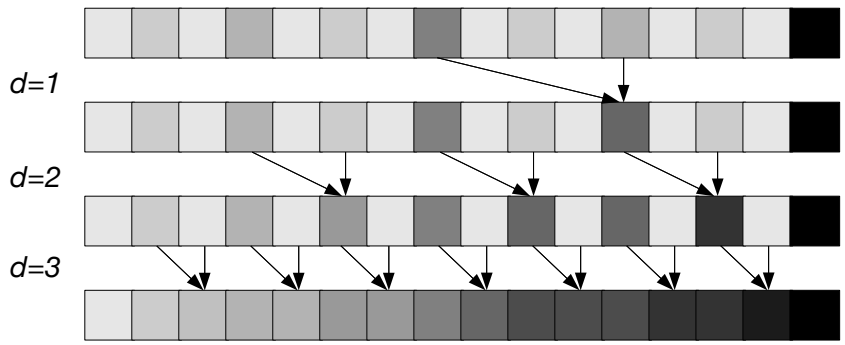


Figure 2: Down-sweep phase of the inclusive scan algorithm

Exclusive down-sweep While the result of the exclusive scan is similar to the result of the inclusive scan, the algorithms are considerably different. In the final result of the exclusive down-sweep phase, each element of the array will contain the sum of all values of its predecessors. Consequently the value of the first element will always be

zero and the value of the last element will not be included in the total sum. The general idea remains the same, as with the inclusive algorithm – the partial sums, computed in the reduction phase, are used to compute new partials sums. However in this version, final results are written in the correct field only in the very last step. Again, the first iteration starts with one active work item, while the others remain idle, and in every subsequent step, the active threads are doubled. Each thread performs one additional task, which is propagating values to preceding elements (depicted by the dotted lines in Figure 3) and hence each thread is represented by three arrows, instead of only two as in the previously introduced phases. This is in contrast to the inclusive scan, where values were only relevant for subsequent elements. By using this property of the algorithm and setting the value of the last element to zero, in the final step it has been propagated to the first element of the array.

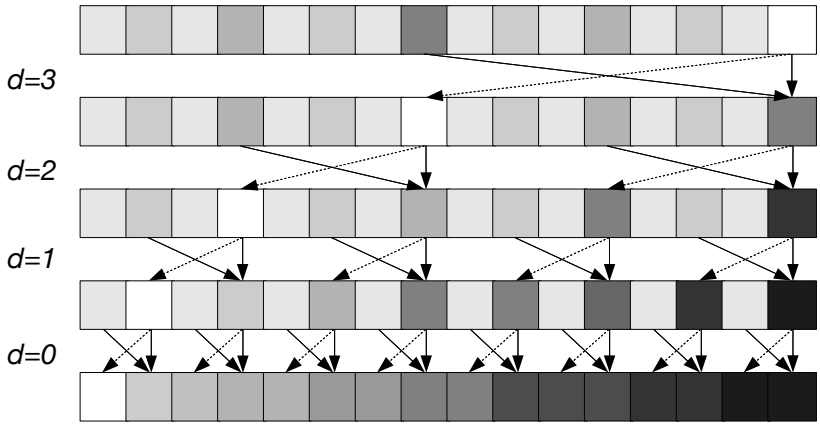


Figure 3: Down-sweep phase of the exclusive scan algorithm

It is clear that, in this version, the scan algorithm can only process arrays with a size of 2^n. A further limitation is the maximum number of work items that the device can run concurrently (maximum size of the local thread group). To overcome the first limitation, an array with a size less than 2^n is padded with zeros to the next highest power of two size. That is, an input array of size 6 is padded to a size of 8 elements, where the last two will be zero. Similarly, an array of size 130 will be padded to 256 elements. Clearly, in certain scenarios, up to approximately 50% of the elements might be padding data, however, since this data will be processed simultaneously, the performance loss will be minimal and probably less, compared to introducing conditional branches, to handle arbitrary sized input in the algorithms themselves. Hence no modifications of

the presented algorithms are required to support input arrays of size not a power of two.

To be able to process large arrays that cannot be processed at once, due to the limited size of the local work group, these arrays need to be split into chunks, where every chunk can be processed by the device. For example, given an array of size 4000 and a maximum local work group size of 256, the array will be split into eight chunks sized 512, since as already mentioned before, an array of size n is processed by $n/2$ work items. The last chunk will be padded by 96 elements. While calculating the number of chunks and their size can be done on the host, the kernels for each phase need to be adjusted. An *offset* variable is introduced that is used when copying elements from global to local memory and back. Its value is the same for all work items in a local group and equivalent to the chunk size times the chunk number. That is, in the given example, threads in the first local group will have an offset of 0, since it's the first chunk they process, threads in the second work group will have an offset of 512, threads in the eight and last work group will have an offset of 3584. Furthermore, the last element of each chunk, which contains its total sum, is copied to a sums array. After all chunks have been processed, an exclusive scan is performed on this array. As a result, each element i will contain the total sum of elements of all chunks, preceding chunk i. In the final step, a kernel is executed, adding this sum to each element in the corresponding chunk. The pseudo code can be found in Listing 5.

```
1  /* compute chunk_size */
2  chunk_size :=nearest_pow2(n >> 1 + n&1)
3  chunk_size := 2*min(chunk_size, max_local_size)
4  n :=nearest_multiple(n, chunk_size)
5
6  for c := 0 to n/chunk_size − 1
7      perform inclusive scan on x[c * chunk_size] to x[(c + 1) * chunk_size − 1]
8      sums[c] := x[(c + 1) * chunk_size − 1]
9
10 perform exclusive scan on sums
11
12 for c := 0 to n/chunk_size − 1
13     for i := 0 to chunk_size do in parallel
14         x[i + c * chunk_size] := sums[c] + x[i + c * chunk_size]
```

Listing 5: Performing a scan on large arrays

Finally, to support very large arrays, another level of intermediate results must be added, since this version of the algorithm is limited by the size of the sums array. That is, for large arrays, the sums array itself needs to be partitioned, its partial sums recorded in another array, which is then scanned, and its values copied to the sums array. For arbitrary large input, even further recursions might be necessary.

The performance of this algorithm is estimated as follows: processing a single chunk requires $O(log_2 n)$ steps for reduction and down-sweep of a chunk. For larger arrays, the scan of the sums array requires $O(log_2 c)$ steps, with c being the number of chunks or $c = n/s_c$ for a chunk size of s_c. Finally, updating all chunks requires $O(c)$. Consequently the total runtime of the algorithm is bound by $O(log_2 n)$. Since it operates in place, $O(n)$ memory is required. Also, the algorithm operates entirely in device memory, meaning that expensive transfers from and to host memory are not required. Furthermore, most of the operations take place in local memory, only having one read and write operation to global memory for each element.

```
1  /* Reduction */
2  for d := 0 to log2n − 1 do
3      for i := 0 to n − 1 do in parallel
4          if i ≥ 2^d
5              x[i] := x[i] + x[i − 2^d]
```

Listing 6: Alternative version of the scan algorithm

There are alternative versions of the parallel scan algorithm, the most common one being the one shown in Listing 6. It has the advantage over the above described version in that the final result is computed in $log_2 n$ steps, which is the same amount of steps the reduction phase requires. However, this version has two drawbacks. Firstly, to process an array of size n, n work items are required. And since the size of a local work group on a device is limited, the chunk size is limited to half the chunk size of the previously described algorithm. Secondly and more importantly, in this version, two threads will try to access the same memory location at the same time. As an example, given an array of 16 elements, in the first step for $d = 0$, thread 1 will read elements 0 and 1, compute the sum and store it again in element 1. At the same time, thread 2 will read elements 1 and 2 and store the result in element 2. In this example, both threads access element 1. Synchronising the access to the colliding memory locations makes little sense, since that would result in a sequential execution of the entire first phase

and furthermore, all threads need to be executed in the correct order (last thread first), to ensure a correct result, yet, this is not guaranteed by OpenCL. To ensure a correct result, the algorithm cannot be performed in place and requires two arrays, which will alternately be used for writing and reading. Hence the choice for the scan algorithm used further on is motivated by the fact that while both versions have the same runtime, the chosen version can process larger chunks, using half the memory.

4.3 Aggregation

In a database system, queries can be deployed that compute an aggregated value of an attribute over a set of tuples. Such can be computing the sum or the average value for this attribute over all tuples, or finding the minimum and maximum values amongst these tuples. All these functions are pretty straightforward: sum will add the value of the respective attributes for all tuples; average will compute the sum and divide it by the number of tuples; minimum will find the lowest value and accordingly, maximum will find the highest value. An example of an average query and its corresponding SQL can be seen in Listing 7.

```
SELECT AVG(t.unique1) AS average
FROM some_table AS t
```

Listing 7: Average query

In the presence of the scan algorithm, described in the previous section, this query is trivial and can be computed as follows. Initially, a kernel will be executed over all tuples, copying the relevant attribute into an array. This array has to be initialised on the host with the size large enough to fit all values and furthermore, comply with the limitations for a scan input array (that is, apply padding if necessary). Once this kernel is finished, a scan will be triggered. However, as the only data we are interested in, is the total sum, as opposed to the partial sums for every element, it is enough to only run the reduction phase, since as already discussed, the final element will contain the total sum. For large arrays that cannot be processed at once, but need to be split into chunks, the total sum can be computed by further reducing the sums array. Once finished, we can compute the average value on the host by reading either the last element of the input array for small arrays, or the last element of the sums array for larger input, and divide it by the total number of tuples that were processed. Memory allocation is also

trivial, since only a single integer needs to be allocated, despite the size of the input.

To compute the sum, only the host application needs to be adjusted and the division by the size of the input must be omitted. To compute the minimum or maximum values, however, a slightly modified version of the scan algorithm is required. Still, only a reduction will be performed, but rather than reading two values, adding them and storing the result into one of the memory locations, each work item needs to compare them and store the smaller or larger value, depending on whether the minimum or maximum is required. Also when padding the array, other values are needed, since 0 could be both the maximum and more likely, the minimum value for a set of tuples. To get the correct result, padded elements need to be initialised with the highest possible number, when computing the minimum and consequently with the lowest possible number, when computing the maximum. Finally, another very common aggregate function in databases is the function `count()`, which computes the total amount of tuples in the result set. For this, an array initialised with 0s must be used, where a kernel can set an element to 1 in case the respective tuple qualifies for the result. Subsequently, the standard scan algorithm can be used, to compute the total number of tuples in the result.

As with the projection query, this query is also limited by the input size. However, in this case, partitioning the input is not enough, since there are dependencies between the different partitions. The correct approach here, is to use an array for the scan, which is large enough to fit all elements, and each partition can then be processed separately, modifying the respective part of that array. Subsequently, it can be processed as described in the previous section.

4.4 Selection

The selection queries are amongst the most common queries that database management systems will perform. They are used to filter a set of tuples, leaving out the one that do not conform to some given criteria. A user will provide these in the form of one or more predicates, which all have to evaluate to true, if a tuple should be included in the result set. Formally, $\sigma_p T$ will return a set of tuples T', such that $\forall t \in T'$ the predicate p is true. Since selection queries are so common, many optimisation techniques have been developed to speed up this process. One of them is maintaining an index that can be used in queries for faster access to certain records. However, there are cases, where the

only available option is to scan the entire table, checking every tuple if it matches the given criteria. This approach is used in this engine, no indexing or other optimisations are employed. The query used in this evaluation has four criteria and can be seen in Listing 8. With the first two, only tuples with a `unique1` value between 1000 and 6000 will be selected. Additionally, tuples that have an odd value for the `unique2` attribute are removed from the input and finally, only these tuples that have the attribute `two` set to zero will be included in the result.

```
SELECT *
FROM some_table AS t
WHERE t.unique1 > 1000
AND t.unique2 % 2 == 0
AND t.two == 0
```

Listing 8: Select query

Generally, such a query can again be parallelised, since every tuple can be examined separately from all other tuples. However, building the result is not as easy as in the previous two examples. There, the output size was known before the execution of the algorithm, hence the correct amount of memory could be allocated beforehand. In the case of projection, the number of tuples and the size of the projected data from a single tuple can be used to compute the required memory and in the case of the aggregation, only a single integer value needs to be allocated. Unfortunately, when performing a selection, the situation is slightly more complicated, since the size of the output is unknown before executing the query. That is, depending on the data and the predicates, with an input size of n tuples, the number of qualifying tuples can be anything between 0 and n. There are two different options on how to deal with this issue. The easier one is to allocate enough memory, to fit the largest result possible, that is allocate the same amount of memory, as the input data uses. Clearly, since in a majority of the cases, not all tuples will match the selection criteria, this approach would result in unused memory. Alternatively, the query can be processed in two steps. In the first one, all matching tuples are found and marked and simultaneously counted. Using their total number, the right amount of memory is allocated before the execution of the second step, where tuples will be copied to the result.

While in the first option, memory is wasted, in the second option, additional computations are required. Due to the fact that the GPU offers more computational power, yet less memory, the second option is chosen for this model. Furthermore, there is another important argument, for avoiding the first option, which will be discussed shortly. As a start, we go into more details on the execution of the kernels. As already mentioned, the query will run in three phases and use two kernels. The first kernel, which can be seen in Listing 9, checks every tuple for the criteria, as defined in the query in Listing 8. It is executed with a global work range of n and local work range of 1, where n is the number of tuples in the input. Local work groups are kept to a size of 1, since no communication between different threads is required at any point. In lines 6 to 8, the relevant attributes are copied from the slower global memory to the faster private memory. Since in the query, the different criteria are connected with a logical AND, all four of them have to evaluate to true, if a tuple has to be copied to the result. This is implemented by means of an integer that will work as a boolean value (even though the OpenCL language supports boolean types, here an integer value is used for consistency with the C standard, where the boolean type is not used). This value is computed by evaluating each criterium and using a bitwise AND operator with its previous value. Finally, the result is stored in a mask field. That is, by the end of this phase, the mask field will contain only 0s and 1s, where a 0 at position x indicates that the tuple at position x in the input array will not be included in the result, whereas a 1 indicates, it will be included. As stated in Section 4.2, the mask array needs to be padded to support the scan algorithm.

This mask array can also be used for determining the total number of tuples in the result, as the sum of all elements (1 for each selected tuple) constitutes the total number of selected tuples. Hence, this can be used to allocate the right amount of memory for the output. For the aggregation query, only a reduction was necessary, to compute the sum of all elements. However, in this query we need to perform a full scan, including an inclusive down-sweep. As a result, the mask array at position x will contain the offset of the respective tuple in the result array. This is a very important aspect, because this way no synchronisation is required for finding the right spot where a certain tuple's place in the result will be. However, since the scan algorithm will populate all fields initially set to 0, a further step is required, since 0s are used to identify tuples that should not be included in the result. This is simply achieved by creating a copy of the

```
1  __kernel void
2  select_init(__global db_tuple * input,
3              __global int * input_mask)
4  {
5          int gid = get_global_id(0);
6          db_attribute_numeric att0 = input[gid].attrb_num[0];
7          db_attribute_numeric att1 = input[gid].attrb_num[1];
8          db_attribute_numeric att2 = input[gid].attrb_num[2];
9
10         int res = att0 > 1000;
11         res &= att0 < 6000;
12         res &= att1 % 2 == 0;
13         res &= att2 == 0;
14
15         input_mask[gid] = res;
16 }
```

Listing 9: Initial selection kernel

mask array before the scan is executed and subsequently, performing a bitwise-OR on all elements, keeping only those elements in the array that were initially set to 1. We use an inclusive scan, so we can identify the first element to be copied to the result. With an exclusive scan, this element would have the value 0, used to mark tuples that are not to be included in the output. Consequently, we need to subtract 1 from the value of the mask, to compute the destination offset in the output mask.

```
1  __kernel void
2  select_finish(__global db_tuple * input,
3                __global int * mask,
4                __global db_tuple * output)
5  {
6          int gid = get_global_id(0);
7          if(mask[gid])
8          {
9                  output[mask[gid]] = input[gid];
10         }
11 }
```

Listing 10: Finishing selection kernel

In the final step, after the result memory has been allocated, another kernel is executed, copying only the relevant data to the output (see Listing 10). This kernel too can be executed with the global range of n and a local work group size of 1. Figure 4 shows an example of how tuples are copied to the output, using this technique: tuples 452 and 454 are copied to the output with offsets 172 and 173 respectively.

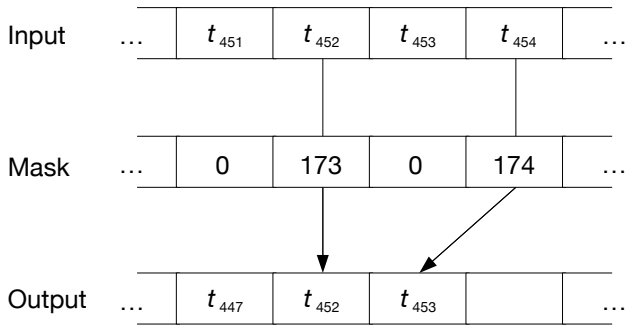


Figure 4: Copying select results using a mask

Alternatively, instead of using two copies of an integer array for the mask, it would be possible to use one array of type integer vector of size two. By doing so, the initialising select kernel would set both values to 1 for each tuple that matches and the scan would only operate on the second out of the two vector values. Finally, the finishing selection kernel can check if the first value is set and if so, use the second value as offset in the result array to write the output. This would save copying the memory and running the extra OR-kernel.

To support any type of predicates (excluding nested queries), only the initialising select kernel needs to be adjusted, loading all relevant data into private memory and performing the accurate checks to match the given criteria. For an input that exceeds the memory limit of the device, a similar approach must be applied, as with the aggregation query, including an additional loop during the final step over the input partitions.

4.5 Hash partitioning

As with the scan algorithm, hash partitioning is not a task that a database management system will perform on its own, but it is part of other algorithms, especially hash joins, which are described in Section 4.6. During this process, a hash table is built, which provides very fast access to a tuple, given a key that can be composed of the values of

one or more of its attributes. Next to finding matching tuples for a join, a hash table can also be used to find and eliminate duplicates.

The advantages of using a hash table over other data structures, is that it is expected that any element in the table can be accessed in $O(1)$. As a tradeoff, hash tables require more memory than other structures and in some cases, construction costs may be higher. The main idea behind hash tables is using a hash function on a key to determine the location of its value. When two keys have the same hash value, different methods exist on where to store their values, the most common of them being chaining, where a list of values is maintained, new ones being added to the end of that list. Such an approach is not suited for parallel processing, as access to a single resource (a list of values) needs synchronisation. An implementation of cuckoo hashing that is optimised for parallel construction and access is described in [Alcantara et al., 2009]. This model uses a very similar approach.

Cuckoo hashing gets its name from the cuckoo bird, which after hatching in another bird's nest, will kick out its step siblings out of it. Similarly, when adding a new key to a table using cuckoo hashing, if there should be another element already occupying the given location, it will be replaced. The algorithm then has to look for another location to store this item. In the basic version, each element belongs to a bucket, where it can be stored in two different locations, but depending on the implementation, the number of the potential locations can be higher. This functionality is achieved by using a different hash function for each location. This model uses the same amount of functions, as the algorithm in [Alcantara et al., 2009], namely three, making it a 3-ary cuckoo hash. Furthermore, each function will put the element in a separate area of the bucket. That is, when using 3 hash functions, the bucket will be split into 3 tables and all elements will have a different location in each of the tables.

Figure 5 shows an example of a bucket, with some elements already added and the consequences of adding another element. In this example, a sequential algorithm is used, based on which the parallel version will be described later. Initially, the bucket contains elements x_0 to x_7, distributed within the three tables. Next, element x_8 is added to the same bucket. However, the location for it collides with the location of element x_3, due to a collision in the hash function with $g_0(x_3) = g_0(x_8) = 1$. As a result, x_3 loses its spot and has to be relocated, adding it to the second table. Due to another collision,

caused by $g_1(x_2) = g_1(x_3) = 2$, element x_2 is ousted by element x_3, which has to be placed in table 3. After another two collisions, element x_6 will be moved to a free spot in table 2 and the insertion process will terminate. In this scenario, the insertion can be completed in five steps. However, in some cases, insertion will not be possible, due to a loop, where the same set of elements will repeatedly replace each other. To avoid this, a counter ensures only a limited number of iterations, after which, new hash functions are generated. Each of them is defined by:

$$g_n(x) = (a_{n0}|seed + (a_{n1}|seed \bmod p)) \bmod s$$

Each function has two predefined constants a_0 and a_1, p is a prime fixed number, s is the size of a bucket table and *seed* is pseudo-random number. To redefine a function, only the seed needs to be changed. Buckets store this number, as it is later used to retrieve elements from it.

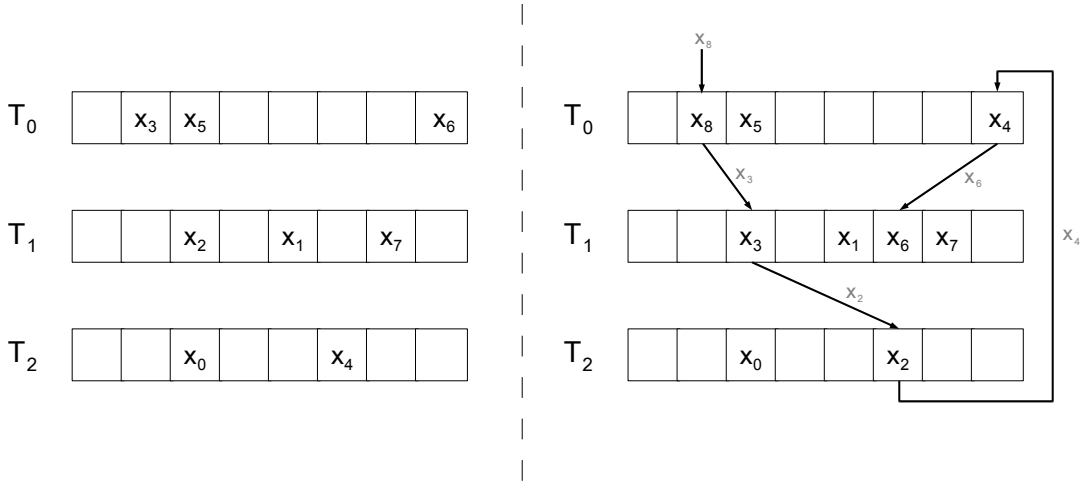


Figure 5: The state of a bucket before and after adding element x_8

In the parallel version of the algorithm, instead of adding one element at a time, a group of n threads will insert n elements simultaneously. When two elements collide and need to be inserted at the same location in the same table, the access to this location will be synchronised by the hardware and one of the threads will eventually succeed over the other. As a result the first element will not have been inserted. To detect this, each thread will check, after writing, if the memory location contains the correct key. In the case it does not, the thread will try again with the next table. Once a thread has managed to store its related element, it will be idle, while those that did not manage try with the next table. Yet in every iteration, all threads in the work group have to check,

if the memory location they used, still contains the element they are responsible for. In this way, once an element gets kicked out by another one, the corresponding thread will know and will then try to find another location. To avoid infinite loops, again a counter must be used and if by the end of it, an element has not been allocated, the bucket seed is changed and the building process starts all over. Once the process is over, the next bucket will be processed. The code of this kernel can be seen in Appendix C.

In the implementation for this model, each bucket can fit a maximum of 512 elements and each of its tables can fit 192 elements, which corresponds to 576 possible locations for elements, leaving at least 64 free spaces. However, since it cannot be guaranteed that exactly 512 elements will fall into each bucket, the number of buckets is derived by assuming a load of 409 element per bucket, which equals approximately 75% average load for buckets and allows deviations of up to 103 more elements. For a uniform distribution, these numbers have been reported to be optimal by [Alcantara et al., 2009] and were hence adopted in this model. While allowing a higher average load factor, will pack elements denser, the time for building the table would increase, as more collisions would occur and rebuilding of entire buckets would also occur with a higher probability. The size of a bucket is limited by the maximum size of a local work group, since all elements in one bucket need to be processed in parallel, to ensure the correctness of the algorithm. As key-value-pairs, the table keeps a pair of the relevant tuple attribute and a reference to the tuple in memory, which keeps the memory overhead very low, compared to keeping a copy of the tuple itself. Data is distributed amongst the buckets, by using the hash function $h(x) = x \bmod n$, where x is the key that will be used for the access and n is the number of buckets.

4.6 Join

In database systems, an equi-join is used to combine tuples from two relations, based on the same value of two attributes. Given two tables T_0 and T_1 with their respective schemas $\mathcal{S}_0$ and $\mathcal{S}_1$, the result R of the join $T_0 \bowtie T_1$ would have the schema $\mathcal{S}_R = \mathcal{S}_0 \cup \mathcal{S}_1$ and contain merged tuples $t_0 \in T_0$ and $t_1 \in T_1$, such that $\pi_a(t_0) = \pi_b(t_1)$, where $a \in \mathcal{S}_0$ and $b \in \mathcal{S}_1$. Hence to compute the result, all matching tuples from both relations need to be found. To do this, several different approaches exist. The easiest one is a nested-loops join. As the name suggests, during the query execution, every tuple from

the first relation is loaded (outer loop) and compared with every tuple of the second relation (inner loop). While nested-loops joins are inevitable for inequi-joins, they are usually avoided for computing equi-joins, due to their bad performance. In another evaluation technique called sort-merge join, both relations are sorted, before their tuples are traversed, which in some cases will greatly increase performance. Finally, another common approach is building a hash table of one of the tables and then iterating over all elements in the other table and checking for matches in the hash table. In many cases, this approach has a better performance than the previous two, and hence is representatively implemented in the model system.

```
SELECT *
FROM some_table AS t1, other_table AS t2
WHERE t1.unique1 = t2.unique1
```

Listing 11: Join query

Listing 11 shows the SQL query that is implemented. With the algorithm for building a hash table present, the actual join query is relatively easy to implement. Once the smaller relation is processed into a hash table, a kernel has to iterate over all tuples in the large relation and check for matching entries in the computed table. A slightly more difficult part is again determining the size of the output, needed to allocate enough memory for the results. Here, a technique, similar to the one used in the select query, will be used. There it was enough to allocate an integer array with the size of the input and set an element to 1, if the tuple that occupies the same position in the input table, qualifies for the result. With the join, there can be no one-to-one mapping between input and mask array, since the input is located in two tables. Therefore, instead of using a simple integer array, this query is taking advantage of one of the built-in vector types in OpenCL. While a vector of size two is required to store references to tuples from either input tables, an extra field is need to store the offset in the result. Hence a vector of size four is employed (the fourth element is used during the scan and will be discussed later in more detail). Furthermore, the attributes of either table are unique, meaning the maximum size of the result is bound by $max(N, M)$, where N and M are number of tuples in each table.

Again, as with the select query, a join will be processed in three steps. In the first phase, a kernel is started that will iterate over all tuples in the larger relation and check

the hash table for matches. Whenever a joining tuple is found, an entry in the output mask is made, setting the first two elements of the vector to 1 and the last two elements to the offset of either joining tuples in their respective table. The kernel is executed with a global range of the size of the larger table, which is also the size of the output mask. Hence, the global ID of a kernel can be used as offset in the mask. After this kernel has finished, an exclusive scan is performed over the first element of the vector array and the total sum is stored. With this knowledge, the right amount of memory for the output is created. Figure 6 shows an example of the contents of the mask array after the scan. Finally, during the last phase, another kernel is executed over the size of the output mask, which will check if the second vector element is set to 1, and if so, will create a merged tuple from the two input tuples, referenced in the last two elements of the vector and write it to the output memory at the location, specified in the first vector field.

As with the previously described queries, this algorithm needs to be extended to support larger inputs that do not fit in device memory. But since partitioning the input is part of the join evaluation algorithm, this task is fairly simple. In case either the larger input table, or the hash table of the smaller input table do not fit on device memory, the following approach needs to be taken. Before partitioning the small table, the algorithm checks, if the input will fit into the device memory. If not, the larger table is partitioned into buckets, without actually building a hash table. Following, a hash table of the smaller input table is built, using the same amount of buckets as the large input requires. Finally, the standard approach can be used, but instead of iterating over the entire input, the algorithm will loop over all buckets, iterating only over tuples from a bucket of the larger input table and checking for matches in the respective bucket of the hash table of the smaller input.

This example of a join algorithm will only work for equi-joins on key-attributes. For a sort-merge join, both tables have to be sorted first (see next section) and subsequently iterated over, to find matching tuples. For a nested-loops join, two techniques can be used, to increase performance. The first one is to use multidimensional work groups. That is, instead of having two loops over each relation, a kernel can be executed within two-dimensional work groups, where the offset in the x-dimension is used to access a tuple from the first relation and the y-dimension is used to access a tuple from the second relation. The other technique consists in first, partitioning both tables in small

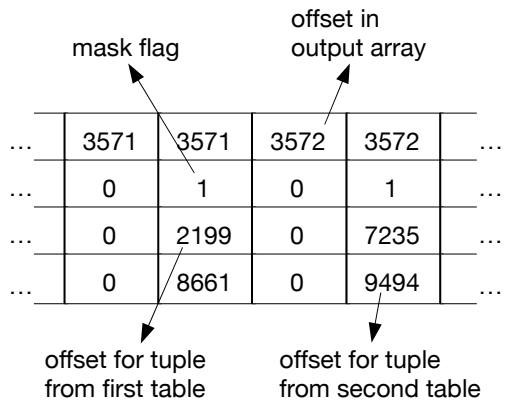


Figure 6: Structure of the mask vector array after the scan

blocks of tuples, and then checking blocks of both tables for matches in parallel. This approach will reduce memory conflicts, as each kernel will operate over a different area of the input. An example for such an algorithm is provided in Listing 12.

```
1  left[n] :=partition(left_table, n)
2  right[n] :=partition(left_table, n)
3  for i := 0 to n − 1 do
4      for j := 0 to n − 1 do in parallel
5          compare_partitions(left[j], right[(j + i) mod n])
6
7  compare_partions(left, right)
8      for each t_L in left
9          for each t_R in right
10             check join predicate
```

Listing 12: Block based nested-loop join

4.7 Sorting

Sorting is a very common problem in computer science and is by far not only limited to the area of databases, where it is used to sort a set of tuples by one or more of their attributes. That is, for a table T sorted by attribute A, for all pairs of tuples t_0 and t_1 in the result, if $t_0 < t_1$, then $\pi_A(t_0) \leq \pi_A(t_1)$. Since the problem of sorting is used in many different scenarios, other than just databases, many parallel sorting algorithms have been proposed. In [Satish et al., 2009] two sorting algorithms optimised for GPUs are introduced, the performance of which exceeds other sorting algorithms so far proposed.

While a sorting operator is not supported by the model system, its implementation does not pose a hurdle. The procedure would be similar to the already discussed queries. In the first step, an array is built that holds the value of the attribute or attributes, based on which the table will be sorted, and a reference to the corresponding tuple. Using one of the proposed algorithms, this array can be sorted. In the final step, a kernel will copy a tuple from a certain position in the sorted array, to the same position in the output. Memory allocation is trivial as well, as the output will have the exact same size as the input relation.

4.8 Combining operators

The queries discussed so far, all implement a single operator (projection, selection, join). However, in a real life database, a query will often contain more than just one operator. Such an example is given in Listing 13, where a join is performed on two relations, which are filtered by some predicates, and finally, an aggregation is performed on the results. The motivation behind this query is to show that it is possible to combine different operators in one query, in a way in that a real live database system would do it. The only thing that has to be done is to execute all operators in the correct order and use the output of each one as input for the next one.

```
SELECT MAX(t.unique1 * t.tenpercent / 100 ) AS average_sum
FROM some_table AS t1, other_table AS t2
WHERE t1.unique1 = t2.unique2
AND t1.unique1 < 6000
AND t2.unique2 > 1000
```

Listing 13: Integrated query

5 Evaluation

In this section, the results of the evaluation of the model system are presented. The main aim is to see how the introduced queries using OpenCL will perform when run on a GPU, compared to a sequential version of the same algorithm, executed by the CPU. Another aspect that is examined is how the input size will affect the processing time and the speedup. This is expected, due to the overhead needed, in preparing and executing each query, caused by compilation of kernels, initialisation of memory buffers, etc., and is likely to have less impact for large input.

To begin with, details on the means used to measure the performance will be given, followed by concrete figures for all queries.

5.1 Setup

For measuring the performance of the parallel versions of the algorithms, each query is evaluated using an NVIDIA GeForce 9600M GT with a clock frequency of 1250 MHz connected to the PCI Express bus with a lane width of 16, and an NVIDIA GeForce 9400M with a clock frequency of 1100 MHz, connected to the PCI bus. Both are equipped with 256 MB of video memory and 16kB of local memory and support 32 bit integer base and extended atomics. For the 9600M GT model, the maximum parallel compute cores are four and maximum amount of work items that can be executed together in a work group is 512. For the 9400M model, the maximum parallel compute cores are two, and it supports the same amount of work items in a work group, as the 9600GT model. The sequential algorithms were executed on a Intel(R) Core(TM)2 Duo CPU with a clock speed of 2660 MHz. It has the same amount of local memory as the GeForce GPUs, however it can directly access the 4GB, 1066 MHz DDR3 main memory. It also supports both base and extended 32 bit integer atomic operations. While the CPU is equipped with two cores, all sequential algorithms only make use of one of them.

Before performing the measurements, the memory throughput of both CPU and GPU is measured. For the CPU, we used the bandwidth tool (see [Zack Smith, 2010]). In a nutshell, the results showed a throughput of approximately 34GB/s for main register to main register transfers and approximately 17GB/s for communication between

stack and registers. The tool also reports 4GB/s throughput of the `memset` function and 1.9GB/s throughput for the `memcpy` function. To measure the bandwidth of the graphics card, GPUBench is used ([Stanford University Graphics Lab, 2004]). The results indicate 9-18GB/sec bandwidth from device memory using various access techniques. Writing data back to main memory is reported to achieve 400-800MB/sec.

As already mentioned earlier, OpenCL in version 1.0 is used. The implementation is the one provided by OS X Snow Leopard.

The run time of the algorithms is measured by reading the time of day before execution and after execution has finished. The difference between the two values is taken as the total time. The same technique is also used for measuring the elapsed time for separate sections within a query. For this, however, explicit flushing of the OpenCL queue is required in certain cases, to ensure the related computation has completed, before reading the elapsed time. Furthermore, as already stated previously, initialisation of the OpenCL context is not taken into account during the measurements, since it is done once at system startup.

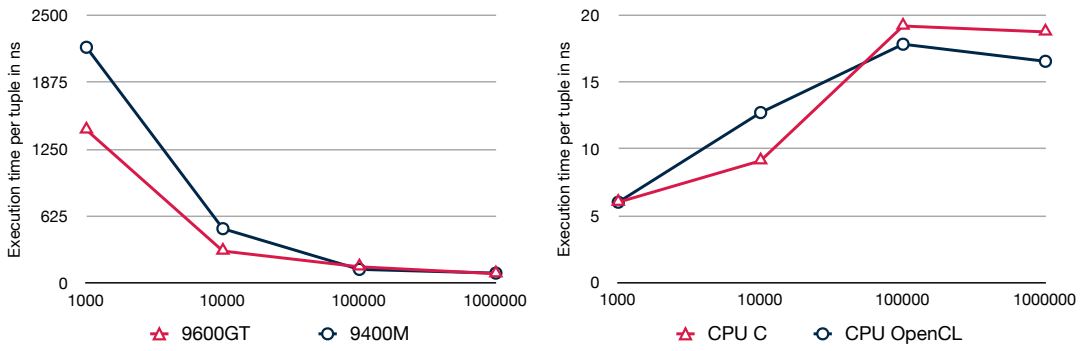
5.2 Results

Each query is executed 100 times in a row, once measuring the total execution time, and once measuring different sections of each query. During each run, the same input is used, which has been loaded into memory at system start up (this time is not included in the measured results). For unary operators, the input is one table object and for binary, it is two table objects. Instead of generating random data each time, files with pre-generated tuples are parsed by the system. These are generated by the data generator, part of the Wisconsin benchmark, as introduced in [DeWitt, 1993]. Using the structure from Appendix A and mapping numeric attributes to integers, decimal attributes to floats, and char attributes to a char array of size 8, each tuple has a size of 80 bytes on the evaluation machine. For measuring how the algorithms scale, the entire evaluation is performed with 1,000, 10,000, 100,000, and 1,000,000 tuples as input for unary algorithms. For binary queries, the first table has the same size, while the size of the second table is defined as $0.6 * N$, where N is the size of the first table. The mean value of all runs is used as a measure, minimising the impact of fluctuations. However, in cases where high fluctuations occur, this is mentioned explicitly.

For comparing the parallel algorithm with a sequential approach, for every query, an algorithm is implemented that uses only one core on the CPU. These algorithms are implemented directly in C and use a similar approach to their parallel counterparts. Most importantly, in both versions the size of the result is determined, before allocating memory for the output. Details on the sequential algorithms are provided in the respective section.

5.2.1 Projection

For evaluating a projection, the query described in Section 4.1 is used. Clearly, this query is not expected to require a high amount of computations, since per tuple, two values are read from, and their sum is written back to global memory. To compare the results with a sequential approach, an algorithm is used that iterates over the input, performing the same operations sequentially with every tuple. Next to running the parallel algorithm on both graphics cards, it is also executed over the CPU. These results are very useful, since they show the overhead of using OpenCL, when compared with the sequential algorithm. It is important to notice that even though the CPU has two cores, the OpenCL implementation does not support local work groups of more than one.



(a) Projection query on the GPU

(b) Projection query on the CPU

Figure 7: Run-time of the projection query per tuple in nanoseconds

Figure 7 shows two graphs, the first depicting the processing time of the GPUs and the second showing the processing time on the CPU. Both graphs show the processing time per tuple in nanoseconds, which is computed by taking the total run time, dividing it by the input size and multiplying it by 10^6. There are two very important observations to make, based on these results. First of all, both approaches show opposite trends,

Tuples	9600GT	9400M	CPU OpenCL	CPU C
1000	1.430	2.202	0.006	0.006
10000	2.980	5.071	0.127	0.091
100000	15.070	12.668	1.781	1.917
1000000	86.200	91.955	16.533	18.730

Table 1: Run-time in ms for the projection query

when increasing the input size. When running the query on a GPU, the processing time decreases substantially with a growing number of input tuples. At the same time, this causes an increase in the time the CPU takes to process a single tuple. The second observation is the immense difference in the execution of the query with the same input when run on the GPU and CPU.

From running the OpenCL code on the CPU, it is clear that the contrasting trends are not due to an overhead in setting up the OpenCL context (compiling kernel, initialising memory objects, etc.), but are a consequence of using a GPU for the query. While for a very high workload, its processing power provides better results, for small workloads the advantage in parallel computing units over the CPU, is lost to the slow memory connection between device and host. This is the cause of the large difference between the total execution times of the GPU and CPU based runs. Even though with the contrasting trends, this difference becomes smaller with a higher number of tuples, for an input of 1 million records, the CPU is still faster that the GPU. The exact numbers can be seen in Table 1. Furthermore, Figure 8 shows the execution times for 1 million tuples of all four runs.

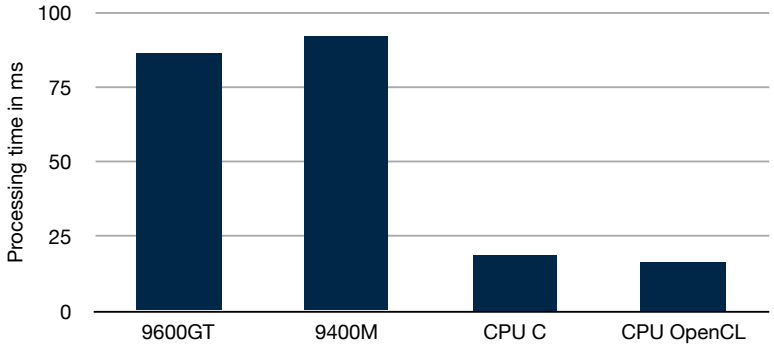
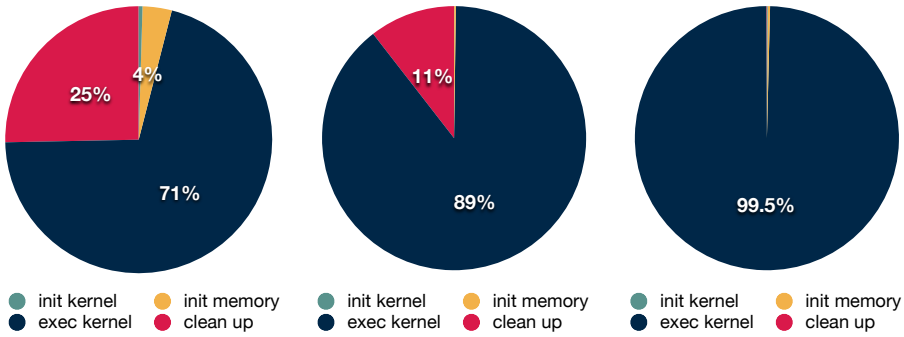


Figure 8: Run-time of the project query for GPU and CPU

In the second part of the projection evaluation, we measured the runtime for four

different sections of the algorithm: (1) compiling the kernel, (2) initialisation of the memory objects, (3) execution of the kernel, and (4) cleaning up memory. In the first section, the only kernel used in this query is compiled. In the second section, a system call is performed to allocate memory for the output, and two memory buffer objects are instantiated, mapping the memory of the input and output memory locations. In the subsequent phase, the kernel execution is triggered, and finally, in the last section the memory buffers and the kernel objects are released. Figure 9 shows the proportions of the different sections for three different runs. Detailed results are provided in Table 5 in Appendix D. The first two were executed on the GPU with 1,000 and 1,000,000 tuples and the last one, on the CPU with 1,000,000 tuples. For all runs, the kernel compilation and initialisation of memory objects is negligibly low, and most importantly, it remains constant, regardless the input size. In contrast, clearing the memory objects takes longer, both for larger input sizes and when executed on GPU, since on the CPU, its execution time can also be neglected. Nevertheless, algorithms spend the majority of time executing the kernel. Since buffer objects are only provided with pointers to the host memory upon initialisation, the actual copying of data is done during the execution of the kernel, rendering it the most time consuming section in the query.



(a) 1,000 tuples / GPU (b) 1,000,000 tuples / GPU (c) 1,000,000 tuples / CPU

Figure 9: Proportions of run-times of different sections of the project query

5.2.2 Aggregation

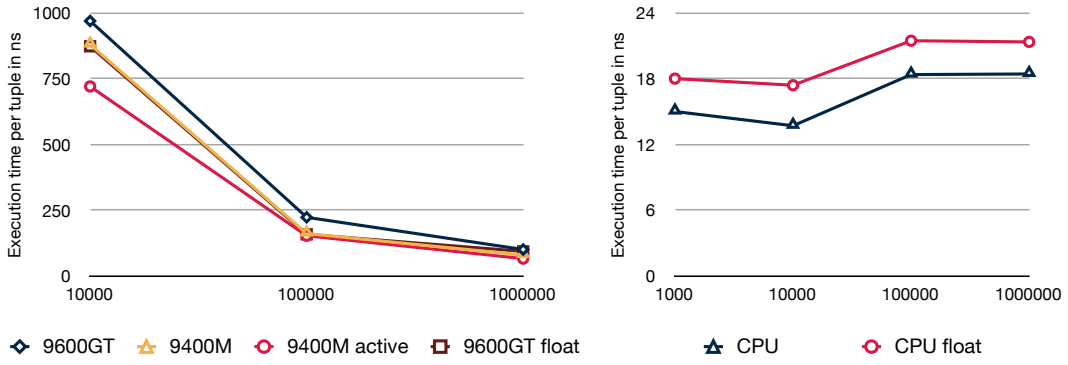
We measured the performance of an aggregation query, by using the query shown in Listing 14, since it has a slightly higher complexity than the query computing the average value, introduced in Section 4.1. For that, a modified kernel is used that copies the given product to an integer array, and subsequently, a modified scan algorithm

is used to compute the maximum value in this array. Instead of computing the sum of two elements and writing it in the respective position, the algorithm evaluates the expression $x * (x >= y) + y * (x < y)$, where x and y are two elements, the result being the maximum of both values. For comparison, a sequential algorithm iterates over all tuples and stores the highest value found so far, using said expression.

```
SELECT MAX(t.unique1*t.tenpercent/100) AS max_value
FROM some_table AS t
```

Listing 14: Max query

This evaluation also provides basic information about the scan algorithm that is used in several other queries as well, however, more details are given in the next section, since this query makes only use of the first phase of the scan algorithm – the reduction. Furthermore, two additional runs are included. The operating system allows to switch between the GeForce 9600GT M for higher performance and the GeForce 9400M for longer battery life. In the previous runs and also, in all other runs to follow, the active card is set to be the GeForce 9600GT M. While in this case it is possible to access both cards in an OpenCL application, if the active card is switched to the 9400M model, the more power consuming 9600GT M is completely deactivated to save energy and hence is not available even in OpenCL programs. The other additional run is a modification of the query, in which instead of storing the maximum value as an integer, it is stored as a float. As a consequence, the scan algorithm also operates over a float array, rather than an integer array. The sequential algorithm is also evaluated twice: once for integers, and once for floats. To compute the value to store in the array, the query evaluates $t.unique1/(0.173 + t.onepercent)$.



(a) on GPUs

(b) on CPU

Figure 10: Run-time per tuple in nanoseconds with different settings

Tuples	9600GT	9400M	CPU
1000	7965	8845	13
10000	1307.3	1451.8	11.5
100000	268.87	208.35	22.05
1000000	146.475	108.544	24.559

Table 2: Run-time per tuple in ns for the selection query

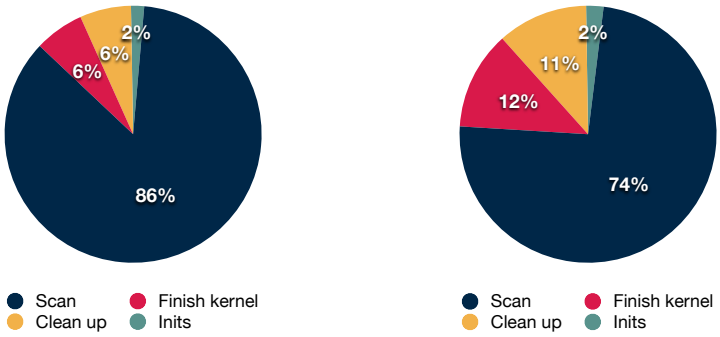
The results can be seen in Figure 10. The trend for the GPU remains the same as with the projection query. The values for 1000 tuples are omitted, to render the difference between the different approaches clearer, but can be seen in Table 6 in Appendix D. There are two interesting facts to note in this diagram. First of all, the best performance is achieved by the, deemed slower, 9400M graphics card, when set as the active card for the system. This is likely to be caused by the driver, as in fact a lower performance would be expected, as when active, the card has to process data from the operating system as well. The other observation is the better performance of the query, when working with floats, instead of with integers. This, however, is not surprising, since the hardware design of GPUs is optimised for floating point computations. Meanwhile, switching from integers to floats leads to a performance degradation on the CPU. Another observation about the CPU is the different trend, compared to the projection query. There, a higher amount of tuples caused higher execution times per tuple, whereas in this scenario, there is no significant difference in the execution time for different input sizes.

5.2.3 Select

For the evaluation of the select operator, the query proposed in 4.4 is used. A sequential version of the algorithm is used, to compare the performance on GPU and CPU. This algorithm also builds a mask, while iterating over all tuples, setting the respective element in the array to 1 in case the tuple should be included in the result. In a second iteration over the mask, tuples are copied from the input to the output. However, no scan over the mask is required, since due to the sequential nature of the algorithm, a simple counter can be used to compute the correct result offset.

The evaluation of this query confirms the results seen so far. As in the projection

query, there is an immense increase in performance of the parallel algorithms on the GPUs, when increasing the input size, while on the CPU, the processing time increases with larger input. For this algorithm, the 9400M model outperforms the faster 9600GT model. The measured times can be seen in Table 2.

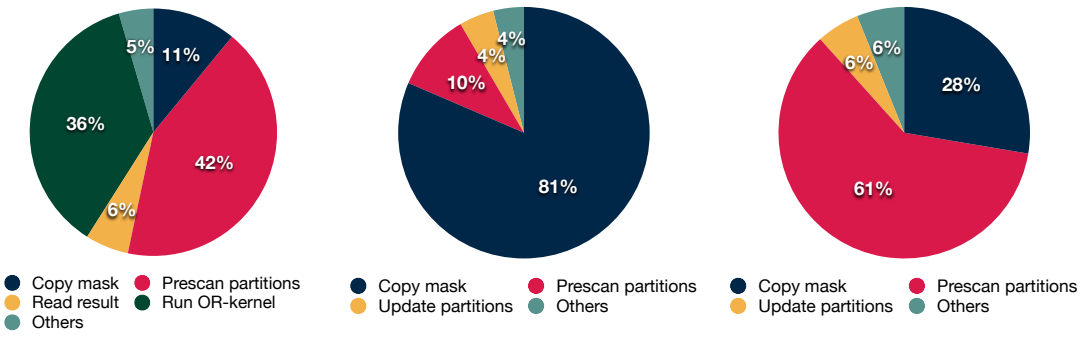


(a) unsing 9600GT (b) using 9400M

Figure 11: Run-time for 1,000,000 tuples of different sections of the select query

The detailed examination of the algorithm, however, bears more interesting results. Figure 11 shows two diagrams, depicting the runtime of the six main sections of the select query on both graphics cards. One would expect the execution of the initialising and finishing kernels to account for the largest part of the total execution time, since they will invoke copying memory from the host to the device and back. This is, however, not the case. As the diagram shows, the query spends most of the time on performing the scan, required to compute the total size of the output and the offset of each element in the result. This is rather unexpected, since during the scan, all data is resident on device memory, and also the algorithm makes use of local memory, avoiding the more costly access to global memory. Only a small part of the execution time is actually spent on the kernels initialising the mask, used as input for the scan, and the kernel, using the output of the scan, to copy the selected tuple to the result. While for smaller input sizes, building the kernels, creating the memory buffers, and cleaning up afterwards takes a notable part of the total execution time, for large inputs, these can be neglected, as they remain more or less constant. The concrete numbers can be seen in Table 8 in Appendix D.

To be able to better understand the delay caused by the scan algorithm, another set of test runs is used to examine in detail its runtime. The select query that was tested uses the simple version of the scan, which operates on a single integer array. And since



(a) 1,000 tuples on 9600GT (b) 1,000,000 tuples on 9600GT (c) 1,000,000 tuples on 9400M

Figure 12: Run-time of different sections of the scan algorithm

the same array is used to determine which tuple should be copied to the result, elements in the array that were initially not set, need to be set to 0, before the scan completes. As described in Section 4.2, this is achieved by creating a copy of the input array and once the inclusive scan completes, the copy is used as an OR-mask, to leave only those elements in the result that were also set in the input. As it turns out, copying the mask is a very costly operation, especially on the 9600GT, where as much as 81% of the total execution time of the scan, is due to this copy operation. Is is also the cause why the 9400M model outperforms the faster 9600GT model. While on the 9400M, the copy operation consumes almost one third of the total run time, the majority of time, the card spends computing the actual prefix sum.

5.2.4 Join

To evaluate the performance of the join algorithm, described in Section 4.6, again a sequential counterpart is implemented. This algorithm follows the same steps, creating a hash table of the smaller input relation, building a result mask while iterating over the large table, and finally build the result using this mask. Again, as with the selection, the mask does not need to be scanned, since a counter can be used for offsetting.

The join algorithm turns out to be the most difficult task for both GPU and CPU. Unlike with the previous algorithms, here the increase in performance on the GPU is not as high, and more importantly, there is a peak in the performance with respect to the input size. That is for a certain input size, both increasing and decreasing it, will lead to performance degradation. The execution time per tuple in nanoseconds can be seen in Table 3. For the GPUs, the fastest processing time is achieved for 100 000 tuples.

Tuples	9600GT	9400M	CPU
1000	15369	13862	447
10000	5249.9	2726.2	180.3
100000	3389.11	1610.9	562.46
1000000	3845.978	2603.181	523.562

Table 3: Run-time per tuple in ns for the join query

For the next higher input of 1 million tuples, the processing time has increased for both graphics card models. Furthermore, for this input size, major fluctuations occur that did not occur in any other scenario. For the 9600GT model, values between 3172.75 to 8877.199 were measured, for the 9400M model, values between 603.348 to 12139.502, and for the CPU, fluctuations between 464.93 and 4597.899 were recorded.

When examining the performance of the algorithm in detail, one can see that the fluctuations are caused by the execution of the partitioning and subsequently the building of the final result. Both phases vary greatly in their execution times. During partitioning, it was said that some buckets might need to be rebuilt, if at the end of the filling stage, there are still elements without a location. These rebuilds are very costly and there are no bounds to the number of times a bucket will have to be rebuilt, before all items are allocated. Furthermore, the larger the input is, the higher the probability is that at least one bucket will have to be rebuilt.

5.2.5 Combined operands

An integrated query uses the already evaluated algorithms, combining them in the following manner: a selection is performed on the larger table; the result is then joined with the smaller table; finally, the max query (adapted for join tuples) is executed with the join result as input. The same applies for both, the GPU based algorithms and the sequential validations. While the trends remain the same, as already introduced (see Table 4), for smaller input sizes the performance on the GPU is considerably worse. However, this can be explained simply by the fact that operations that take a constant, small amount of time (such as compiling kernels and initialising memory buffers), have a higher impact for small tuples. And since the integrated query combines three of the queries, the overhead of each one of them has a negative effect on the processing

Tuples	9600GT	9400M	CPU
1000	20717	25466	333
10000	3452.9	3847.8	59.8
100000	679.60	644.11	78.35
1000000	326.175	233.390	76.022

Table 4: Run-time per tuple in ns for the integrated query

time. Furthermore, this query is naturally more time consuming, since it combines the complexity of all the other queries.

5.2.6 Varying the tuple size

As an additional test, the max query of Section 5.2.2 was evaluated in three different runs, using different tuple sizes, each consisting of a different amount of attributes. Small attributes have a tuple ID, two attributes of type integer, one attribute of type float, and one attribute of type char, resulting in a size of 24 bytes on the machine used for the evaluation. Large tuples have the same structure as the standard sized tuples, previously introduced, but instead of integer attributes, they have long integer attributes, resulting in a size of 136 bytes. Tuples of all sizes use float values for numeric attributes, since double is optional and not guaranteed by OpenCL. By having tuples of various size, the performance of the system can be examined under different memory-computation ratios. That is, using large tuples will result in a higher memory bandwidth per tuple, compared to small tuples, while the computation costs will remain the same. This is due to the fact that the query always uses the same attributes, present in all types of tuples, regardless their size.

However, as depicted by Figure 13, changing the size of a tuple does not have a major impact on any of the algorithms. While the processing time increases with the size of the tuple, this does not correspond to the increase in size. That is, while a large tuple is more than five times the size of the small tuple, the run-time for the former is approximately 25% higher, than the time required to process a large tuple.

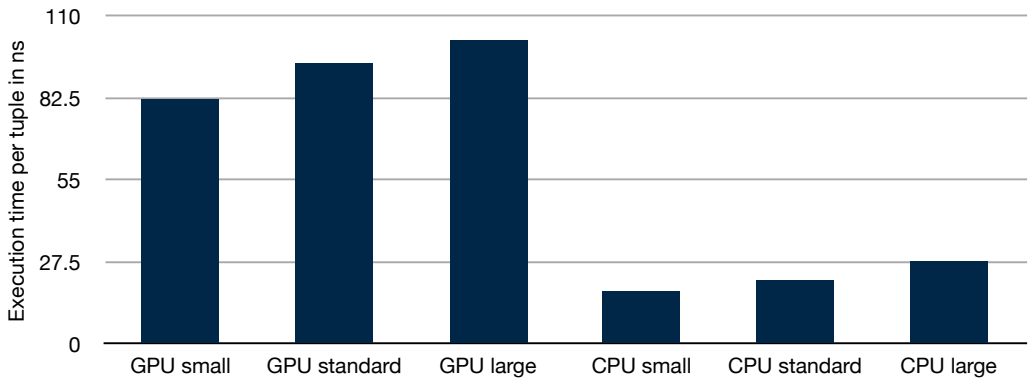


Figure 13: Influence of the tuple size on the performance

5.2.7 Reading and writing data

The results we have seen so far, show a low performance of the GPU for small inputs and also in certain cases, a large amount of time is spent on copying data. To get a clearer picture, a simple program is run, to measure the memory throughput for reading and writing different amounts of data from host memory to device memory and back. To achieve this, the input data is copied to the device, using a blocking copy operation, followed by a blocking reading operation. By measuring the time for either operation with varying input size, we can compute the achieved throughput for the given memory size. The results of this experiment can be seen in Figure 14. It is clear that while theoretically a high bandwidth is provided by the bus, for small amounts of data this limit cannot be reached.

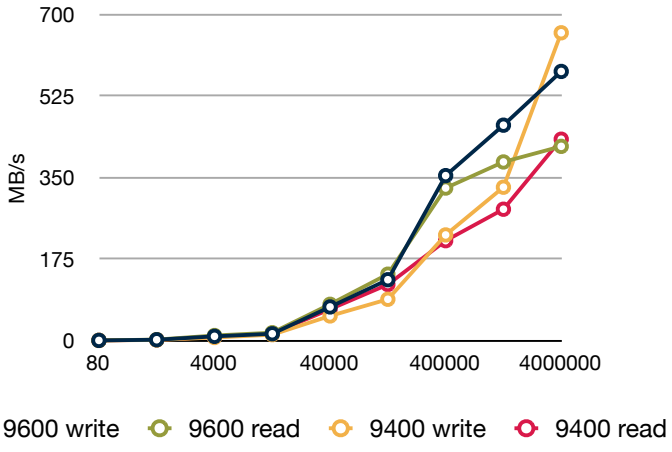


Figure 14: Memory throughput for increasing amount of data

6 Related Work

Many papers have been published on increasing database performance by parallelising the processing algorithms in the context of parallel databases ([DeWitt and Gray, 1992] and [DeWitt et al., 1986]). Consequently the nature of the problems that arise with database systems running on different machines is not necessarily the same as the ones that arise on a single multi-core machine. Parallel databases employ a shared-nothing architecture, where each site operates on its own data, using its own memory and processing resources. Such a system is described in detail in [DeWitt et al., 1986]. It has a number of interconnected nodes where each node consists of a hard drive where a portion of the horizontally partitioned data can be found and a processor that will perform tasks on the site's data portion. The authors discuss the implementation of various algorithms, such as selection and joining. When operations need to be performed on data from multiple sites, the main technique used to maintain a high level of parallelism is hashing. An important aspect of the proposed prototype is the communication between the different sites. However, in the model discussed in this thesis, communication can happen by exchanging data in main memory, hence the discussed communication schemes are not relevant.

A more closely related approach is introduced in [Krikellas et al., 2010]. The authors discuss the parallelising of query evaluation algorithms on multi-core processors. The difference between the two approaches lies in the fact that one is fine-tuned for a multi-core CPU with its respective cache hierarchies, whereas the model proposed here is kept relatively generic, since being based on OpenCL, it has to run on different kinds of devices, each with a different memory architecture.

A good amount of work has been published, examining the general performance of GPGPUs ([Lee et al., 2010], [Owens et al., 2007]). Furthermore, papers on specific topics have been published, showing how to use GPUs to increase the performance of commonly used algorithms ([Satish et al., 2009], [Alcantara et al., 2009]). However, no work has yet been published examining the performance of a data-intensive application, such as a database management system, when using GPGPUs.

7 Conclusion and Future Work

The purpose of this thesis was to evaluate the performance of a database system, evaluating queries using OpenCL. GPGPUs are deemed the future of performance programming. Unfortunately, as the results provided herein show, graphics cards are not suitable for such a high memory bandwidth application, such as a database system. The main bottleneck is the low memory throughput between host and device for small amounts of data, as well as the limited amount of available device memory. Since in many cases, the computational effort for a single tuple is relatively small, no significant gains can be achieved, by using the many computational cores of the GPU, when in the time that data is copied to the device, a common CPU will be able to complete the task. However, this is the case for small- to middle-sized input tables. As the trends showed, the larger the input, the better performance can be achieved on the GPU, while there is no increase, but often decrease in the performance of a CPU approach on the same input. It is also important to keep in mind that the evaluation was performed using middle class graphics cards. Modern hardware is expected to perform better in some aspects, however, the speed with which data can be copied to and from the device, is unlikely to increase in the near future in such an extent, to make their use for databases of all sorts reasonable. Also, while more expensive graphics cards will come equipped with more device memory, most of them still have a very limited amount of local memory, which cannot be used by queries.

Another aspect, limiting the performance and applicability of this approach, is the dependency on device drivers and the OpenCL implementation. Both can have a major impact on the performance. For a strict CPU approach, such dependencies do not exist. As an example, [Karimi et al., 2010] introduces a performance comparison, showing that applications running CUDA will perform better than the same applications, running OpenCL. Nevertheless, running the projection query on the CPU using OpenCL, showed a performance increase, indicating that the reason for the bad performance lies indeed with the use of the graphics cards, rather than with using OpenCL.

Future work This work provides only a basic insight in the problem that is performing database queries on GPGPUs. All the algorithms that were introduced, though

designed to profit from parallelism, still can be optimised in many ways. To begin with, all of them were planned to perform well for large input. The results showed that, indeed, performance increases for larger inputs, but is very bad for small input. Another way to increase performance is to use special addressing within kernels to avoid bank conflicts. Further optimisation information is available from [NVIDIA, 2009].

This model is also limited to a single query at a time. However, any database system needs to support execution of multiple queries simultaneously. While this can be easily achieved, by using the task parallelism support of OpenCL (as of OpenCL 1.1 even multiple devices can be used simultaneously), it is interesting to investigate how this would affect performance, as there will be even less device memory available for each query, since they all need to share the global memory pool. However, at the same time the cores could be kept busier. In general, other, computationally more complex scenarios can be evaluated, such as query evaluation over XML documents.

Encountered problems and difficulties Working with OpenCL and parallel programming in general required a certain time for adjusting from working with sequential programs. While the OpenCL API is intuitive and very well documented, it is still hard to develop kernels, as no means of debugging are available. Since the code is executed on the GPU, the use of a debugger is not possible. Furthermore, no output messages can be used. The only option is to provide additional objects, in which to store debugging information, write them back to host memory and print their contents. Due to the queue being asynchronous, some errors only occur at a later time and it is not always easy to find the cause for them. As an example, during the evaluation, a memory leak (a memory buffer not being released), invalidated the command queue, but only after several iterations.

While it is possible to use custom objects (structs) in kernels, these have to be defined in the same program. Includes, as used in standard C and in other languages, are not supported. As a consequence, each time a structure or special function needs to be used in a kernel, the program, containing the latter must define them. In the case of type definitions, two versions of the code need to be maintained, one for the host application, and one within the program, defining the kernel to use them.

Finally, for structures that are created by multiple mallocs, every separate memory

block needs to be passed explicitly to the device application (that is copy the memory to a buffer and pass it as argument to a kernel). For example, for building a hash table, first the hash table object itself is created, by allocating the necessary amount of memory. Subsequently, a various amount of buckets will be created, depending on the size of input that will be mapped onto the table. To use the hash table in a kernel, two memory objects need to be created and passed as arguments to the kernel. The first will contain the hash table with its meta data, and the second will contain the buckets.

A Table structure, as used by the Wisconsin benchmark

```
CREATE TABLE ‘table_name‘ (
        ‘unique1‘ NUMERIC,
        ‘unique2‘ NUMERIC,
        ‘two‘ NUMERIC,
        ‘four‘ NUMERIC,
        ‘ten‘ NUMERIC,
        ‘twenty‘ NUMERIC,
        ‘onepercent‘ DECIMAL,
        ‘tenpercent‘ DECIMAL,
        ‘twentypercent‘ DECIMAL,
        ‘fiftypercent‘ DECIMAL,
        ‘unique3‘ NUMERIC,
        ‘even‘ NUMERIC,
        ‘odd‘ NUMERIC,
        ‘stringu1‘ CHAR(8),
        ‘stringu2‘ CHAR(8),
        ‘stringu4‘ CHAR(8)
);
```

B Running a projection in OpenCL

```
db_attribute_numeric * projection ( openCL * cl_ctx, db_table * input )
{
    cl_kernel kernel = create_kernel(&cl_ctx->programs[K_PROJECTION], "sum");
    db_attribute_numeric * res = calloc(input->row_count, sizeof(db_attribute_numeric));

    cl_mem source = clCreateBuffer(cl_ctx->ctx,
                        CL_MEM_READ_ONLY | CL_MEM_USE_HOST_PTR,
                        input->row_count * sizeof(db_tuple), (void*) input->tuples, NULL);
    cl_mem destination = clCreateBuffer(cl_ctx->ctx,
                        CL_MEM_WRITE_ONLY | CL_MEM_USE_HOST_PTR,
                        input->row_count * sizeof(db_attribute_numeric), (void*) res, NULL);

    clSetKernelArg(kernel, 0, sizeof(cl_mem), &source);
    clSetKernelArg(kernel, 1, sizeof(cl_mem), &destination);

    size_t global_work_size = input->row_count;
    size_t local_work_size = 1;
    clEnqueueNDRangeKernel(cl_ctx->queue, kernel, 1, NULL,
                        &global_work_size, &local_work_size, 0, NULL, NULL);
    clFinish(cl_ctx->queue);

    clear_kernels(1, kernel);
    clear_memory(2, source, destination);

    return res;
}
```

C Kernel for building a bucket of the hash table in parallel

```
__kernel void
process_bucket ( __global db_tuple * input,
                 __global int * partitions,
                 __global hash_table_bucket * buckets,
                 __local hash_table_temp_bucket * bucket,
                 const int bucket_no)
{
        int lid = get_local_id(0);
        int item = partitions[bucket_no*512+lid];
        int key = input[item].attrb_num[0];

        int table = 0, hash = 0, set = 0;

        do
        {
                if(lid==0) bucket->done = 1;
                int seed = buckets[bucket_no].seed;
                set = 0;

                for(int i = 0; i < 25; i++)
                {
                        if(!set)
                        {
                                hash = g(key, seed, table);
                                bucket->tables[table].entry[hash] = key;
                                set = 1;
                        }
                        barrier(CLK_LOCAL_MEM_FENCE);

                        if(bucket->tables[table].entry[hash] != key)
                        {
                                table = (table+1) % 3;
                                set = 0;
                        }
                }
```

```
                if(bucket->tables[table].entry[hash] != key)
                {
                        buckets[bucket_no].seed++;
                        bucket->done = 0;
                }
                barrier(CLK_LOCAL_MEM_FENCE);

        } while(!bucket->done);

        // copy local to global
        buckets[bucket_no].tables[table].entry[hash].key = key;
        buckets[bucket_no].tables[table].entry[hash].value = item;
}
```

D Evaluation results

Device	Input	Total	Init kernel	Init memory	Exec kernel	Clean up
9600GT	1000	1.530	0.007	0.054	1.078	0.386
9600GT	10000	2.306	0.004	0.061	1.831	0.405
9600GT	100000	13.135	0.010	0.115	11.420	1.582
9600GT	1000000	84.519	0.009	0.173	75.446	8.883
9400M	1000	1.582	0.006	0.053	1.136	0.381
9400M	10000	2.955	0.006	0.085	2.432	0.427
9400M	100000	13.830	0.011	0.164	12.312	1.334
9400M	1000000	84.378	0.009	0.163	74.934	9.265
CPU	1000	0.054	0.006	0.007	0.031	0.005
CPU	10000	0.139	0.006	0.035	0.089	0.004
CPU	100000	1.818	0.017	0.025	1.758	0.013
CPU	1000000	16.626	0.015	0.049	16.545	0.012

Table 5: Run-time in ms for the projection query and its 4 main sections

Input	9600GT	9400M	9400M active	9600GT float	CPU	CPU float
1000	4545	5367	3865	4382	15	18
10000	970.2	880.4	721.2	874.0	13.7	17.4
100000	223.61	161.74	153.39	158.82	18.37	21.46
1000000	101.183	80.069	66.426	94.018	18.427	21.351

Table 6: Run-time in ms for the aggregation query with different settings

Device	Input	small	standard	large
9600GT	1000	4402	4407	4458
9600GT	10000	876.8	863.2	913.7
9600GT	100000	172.43	160.03	169.73
9600GT	1000000	81.934	94.015	101.803
CPU	1000	18	18	19
CPU	10000	17.0	17.6	17.8
CPU	100000	17.02	21.30	27.68
CPU	1000000	17.614	21.204	27.660

Table 7: Run-time in ms for the aggregation query with different tuple sizes

Device	Section	1000	10000	100000	1000000
9400	Total	8.830	14.599	20.887	105.498
9400	Init kernel	0.005	0.005	0.007	0.010
9400	Init memory	0.062	0.056	0.106	0.145
9400	Exec init kernel	0.949	0.962	0.917	2.024
9400	Scan	5.963	11.084	15.141	78.088
9400	Init result set	0.001	0.002	0.002	0.097
9400	Init result memory	0.008	0.032	0.031	0.028
9400	Exec finish kernel	1.160	1.712	2.836	13.073
9400	Clean up	0.623	0.737	1.837	12.023
9600	Init kernel	7.997	12.779	27.749	146.461
9600	Init memory	0.004	0.004	0.008	0.009
9600	Exec init kernel	0.047	0.049	0.117	0.158
9600	Scan	0.858	0.845	0.882	2.079
9600	Init result set	5.626	10.271	22.733	125.443
9600	Init result memory	0.002	0.002	0.002	0.131
9600	Exec finish kernel	0.018	0.019	0.068	0.045
9600	Clean up	0.938	1.020	2.451	9.130
9600	Clean up	0.495	0.560	1.479	9.457

Table 8: Run-time in ms for the different sections in the select query

References

[Alcantara et al., 2009] Alcantara, D., Sharf, A., Abbasinejad, F., Sengupta, S., Mitzenmacher, M., Owens, J., and Amenta, N. (2009). Real-time parallel hashing on the GPU. *ACM Transactions on Graphics (TOG)*, 28(5):1–9.

[AMD, 2010] AMD (2010). Advanced Micro Devices, Inc. `http://www.amd.com/`.

[Apple, 2009] Apple (2009). OpenCL Parallel Prefix Sum (aka Scan) Example. `http://developer.apple.com/mac/library/samplecode/OpenCL_Parallel_Prefix_Sum_Example/Introduction/Intro.html`.

[Apple Inc., 2010] Apple Inc. (2010). `http://www.apple.com/`.

[Core 2 Duo, 2010] Core 2 Duo (2010). Intel® Core™2 Duo Processor E8000 and E7000 Series. `http://download.intel.com/design/processor/datashts/318732.pdf`. Datasheet.

[CUDA, 2010] CUDA (2010). Compute Unified Device Architecture. `http://www.nvidia.com/object/cuda_home_new.html`. NVIDIA.

[DeWitt, 1993] DeWitt, D. (1993). The Wisconsin benchmark: Past, present, and future. *The Benchmark Handbook*.

[DeWitt et al., 1986] DeWitt, D. J., Gerber, R. H., Graefe, G., Heytens, M. L., Kumar, K. B., and Muralikrishna, M. (1986). Gamma - a high performance dataflow database machine. In *VLDB*, pages 228–237.

[DeWitt and Gray, 1992] DeWitt, D. J. and Gray, J. (1992). Parallel database systems: The future of high performance database systems. *Commun. ACM*, 35(6):85–98.

[GeForce9600GT, 2008] GeForce9600GT (2008). NVIDIA GeForce 9600 GT. `http://www.nvidia.com/object/product_geforce_9600gt_us.html`. GeForce 9600 GT.

[Guide, 2010] Guide, P. (2010). Intel® 64 and IA-32 Architectures Software Developer's Manual.

[Harris, 2007] Harris, M. (2007). *Parallel Prefix Sum (Scan) with CUDA*. NVIDIA.

[Intel, 2010] Intel (2010). Intel® Corporation. `http://www.intel.com/`.

[Karimi et al., 2010] Karimi, K., Dickson, N. G., and Hamze, F. (2010). A performance comparison of CUDA and OpenCL. *CoRR*, abs/1005.2581.

[Khoronos Group, 2010] Khoronos Group (2010). `http://www.khronos.org/`.

[Kilgariff and Fernando, 2005] Kilgariff, E. and Fernando, R. (2005). The GeForce 6 Series GPU Architecture. *GPU Gems*, 2:471–491. `http://http.download.nvidia.com/developer/GPU_Gems_2/GPU_Gems2_ch30.pdf`.

[Krikellas et al., 2010] Krikellas, K., D. Viglas, S., and Cintra, M. (2010). Modeling Multithreaded Query Execution on Chip Multiprocessors. In *Accelerating Data Management Systems Using Modern Processor and Storage Architectures (ADMS)*.

[Lee et al., 2010] Lee, V. W., Kim, C., Chhugani, J., Deisher, M., Kim, D., Nguyen, A. D., Satish, N., Smelyanskiy, M., Chennupaty, S., Hammarlund, P., Singhal, R., and Dubey, P. (2010). Debunking the 100x gpu vs. cpu myth: an evaluation of throughput computing on cpu and gpu. *SIGARCH Comput. Archit. News*, 38(3):451–460.

[NVIDIA, 2009] NVIDIA (2009). NVIDIA OpenCL Best Practices Guide. `http://www.nvidia.com/content/cudazone/CUDABrowser/downloads/papers/NVIDIA_OpenCL_BestPracticesGuide.pdf`.

[NVIDIA Corporation, 2010] NVIDIA Corporation (2010). `http://www.nvidia.com/`.

[OpenCL 1.0 Specification, 2009] OpenCL 1.0 Specification (2009). OpenCL 1.0 Specification. `http://www.khronos.org/registry/cl/specs/opencl-1.0.48.pdf`. Khronos Group.

[Owens et al., 2007] Owens, J., Luebke, D., Govindaraju, N., Harris, M., Kr "uger, J., Lefohn, A., and Purcell, T. (2007). A survey of general-purpose computation on graphics hardware. In *Computer Graphics Forum*, volume 26, pages 80–113. John Wiley & Sons.

[Ramanathan et al., 2006] Ramanathan, R., Contributors, P., Curry, R., Chennupaty, S., Cross, R., Kuo, S., and Buxton, M. (2006). Extending the world's most popular processor architecture. *Technology@ Intel Magazine*.

[Satish et al., 2009] Satish, N., Harris, M., and Garland, M. (2009). Designing efficient sorting algorithms for manycore gpus. *Parallel and Distributed Processing Symposium, International*, 0:1–10.

[Stanford University Graphics Lab, 2004] Stanford University Graphics Lab (2004). GPUBench. http://graphics.stanford.edu/projects/gpubench/.

[Zack Smith, 2010] Zack Smith (2010). Bandwidth memory benchmark. http://home.comcast.net/~fbui/bandwidth.html. Bandwidth benchmark.