TECHNISCHE UNIVERSITÄT ILMENAU
Institut für Praktische Informatik und Medieninformatik
Fakultät für Informatik und Automatisierung
Fachgebiet Datenbanken und Informationssysteme

Dissertation

# Finding the Right Processor for the Job
—
# Co-Processors in a DBMS

vorgelegt von

Dipl.-Inf. Hannes Rauhe

geboren am 5.9.1985 in Meiningen

zur Erlangung des akademischen Grades

Dr.-Ing.

1. Gutachter: Prof. Dr.-Ing. habil. Kai-Uwe Sattler

2. Gutachter: Prof. Dr.-Ing. Wolfgang Lehner

3. Gutachter: Prof. Dr. Guido Moerkotte

urn:nbn:de:gbv:ilm1-2014000240

Ilmenau, den 19. Oktober 2014

# Abstract

Today, more and more Database Management Systems (DBMSs) keep the data completely in memory during processing or even store the entire database there for fast access. In such system more algorithms are limited by the capacity of the processor, because the bottleneck of Input/Output (I/O) to disk vanished. At the same time Graphics Processing Units (GPUs) have exceeded the Central Processing Unit (CPU) in terms of processing power. Research has shown that they can be used not only for graphic processing but also to solve problems of other domains. However, not every algorithm can be ported to the GPU's architecture with benefit. First, algorithms have to be adapted to allow for parallel processing in a Single Instruction Multiple Data (SIMD) style. Second, there is a transfer bottleneck because high performance GPUs are connected via PCI-Express (PCIe) bus.

In this work we explore which tasks can be offloaded to the GPU with benefit. We show, that query optimization, query execution and application logic can be sped up under certain circumstances, but also that not every task is suitable for offloading. By giving a detailed description, implementation, and evaluation of four different examples, we explain how suitable tasks can be identified and ported.

Nevertheless, if there is not enough data to distribute a task over all available cores on the GPU it makes no sense to use it. Also, if the input data or the data generate during processing does not fit into the GPU's memory, it is likely that the CPU produces a result faster. Hence, the decision which processing unit to use has to be made at run-time. It is depending on the available implementations, the hardware, the input parameters and the input data. We present a self-tuning approach that continually learns which device to use and automatically chooses the right one for every execution.

# Abbreviations

**AES** Advanced Encryption Standard

**AVX** Advanced Vector Extensions

**BAT** Binary Association Table

**BLAS** Basic Linear Algebra Subprograms

**CPU** Central Processing Unit

**CSV** Comma Separated Values

**DBMS** Database Management System

**DCM** Decomposed Storage Model

**ETL** Extract, Transform, Load

**FLOPS** Floating Point Operations per Second

**FPGA** Field-Programmable Gate Array

**FPU** Floating Processing Unit

**GPU** Graphics Processing Unit

**GPGPU** General Purpose Graphics Processing Unit

**HDD** Hard Disk Drive

**I/O** Input/Output

**JIT** Just-In-Time

**LWC** Light-Weight Compression

**MMDBMS** Main Memory Database System

**ME** Maximum Entropy

**MIC** Many Integrated Core

**MVS** Multivariant Statistics

**NOP** No Operation

**ODBC** Open Database Connectivity

**OLTP** Online Transactional Processing

**OLAP** Online Analytical Processing

**ONC** Open-Next-Close

**OS** Operating System

**PCIe** PCI-Express

**PPE** PowerPC Processing Element

**QEP** Query Execution Plan

**QPI** Intel QuickPath Interconnect

**RAM** Random Access Memory

**RDBMS** Relational Database Management System

**RDMA** Remote Direct Memory Access

**SIMD** Single Instruction Multiple Data

**SMX** Streaming Multi Processor

**SPE** Synergistic Processing Element

**SQL** Structured Query Language

**SSD** Solid State Drive

**SSE** Streaming SIMD Extensions

**STL** Standard Template Library

**TBB** Intel's Thread Building Blocks

**UDF** User-Defined Function

**UVA** Universal Virtual Addressing

# Contents

# 1. Introduction

In the 1970s the idea of a database machine was a trending topic in research. The hardware of these machines was supposed to be built from ground up to serve only one purpose: efficiently accessing and processing data stored in a Database Management System (DBMS). One of the common ideas of the different designs proposed was the usage of a high number of processing units in parallel. The evolution of the Central Processing Unit (CPU) and disks overtook the development of these database machines and in the 80s, the idea was declared a failure [11]. Thirty years later researchers again proposed to use a massively parallel architecture for data processing, but this time, it was already available: the Graphics Processing Unit (GPU). However, except for some research systems there is still no major DBMS that runs on GPUs.

We think that the reason for this is the versatility of modern DBMS. An architecture for DBMS must be able to do every task possible reasonably well, whether it is aggregating data, simple filtering, processing transactions, complex mathematical operations, or the collection and evaluation of statistics on the data stored in the system. A specialized processor may be able to do a subset of these tasks very fast, but then it will fail to do the rest. The GPU is able to process a huge amount of data in parallel, but it has problems with short transactions, which are not suitable for parallel processing, because they require a consistent view on the data.

However, nowadays different processing units are available in one system. In recent research therefore the focus switched to the usage of heterogeneous hardware and the concept of co-processing. Instead of calculating all by itself, the CPU orchestrates a number of different processing units within the system and decides for each task, where to execute it. The question therefore not longer is: "How does the perfect processor for a DBMS look?", but
*"Which of the available processor is the right one to use for a certain task?"*.

## 1.1. Motivation

On the one hand there are many different tasks a DBMS has to process to store data and keep it accessible. There are the operators used for query execution, such as selection, join, and aggregation with simple or complex predicates. Under the hood the systems executes much more logic to maintain stored data and collect meta data on how users access the content. These statistics are used to optimize the query execution; partly with complex mathematical methods. Additionally, most vendors position their DBMS more and more as a data management platform that is not only able to execute queries but any application logic defined by the user as well.

On the other hand every standard PC can be equipped with powerful co-processors, specialized on a certain domain of algorithms. Almost any modern PC already has a GPU which is able to solve general purpose algorithms with the help of its massively parallel architecture. Additionally, there are Field-Programmable Gate Arrays (FPGAs), Intel's Xeon Phi, or IBM's Cell processor. Of course, any task can be solved by the CPU, but since its purpose is so generic, other hardware may be

- cheaper,
- more efficient in terms of memory consumption,
- or simply faster

at executing the task. Even if the co-processor is just as good as the CPU for a certain problem; if it is available in the system anyhow it can be used to free resources on the CPU for other jobs.

However, in contrast to co-processors that are integrated into the CPU—such as the Floating Processing Unit (FPU) or cryptography processors, e.g., for accelerating Advanced Encryption Standard (AES)–the co-processors mentioned in the last paragraph cannot be used by just re-compiling the code with special instructions. Instead, they require their own programming model and special compilers. That means that most algorithms have to be re-written to run on co-processors. This re-write is not trivial, because especially GPU algorithms require a completely different approach. Since the prediction of the performance of those new algorithm is impossible due to the complexity of the hardware, there are three questions to answer:

- Which tasks can be ported to a co-processor and how is this done?
- Which tasks are likely and which are unlikely to benefit from another architecture?
- How can the system automatically decide on the best processing unit to use for a specific task?

At the moment, the data to be processed by a task usually has to be copied to the co-processor, so data-intensive tasks are usually unlikely to benefit from the GPU.

However, since we reached a peak of single-thread CPU performance because of physics [57], CPU vendors are changing their architectures to support for instance vectorized and multi-threaded execution. These concepts are already built to an extreme in GPUs and FPGAs. In heterogeneous architectures, GPUs are integrated into the CPU, e.g., AMD's Fusion architecture [14]. Hence, modifications to simple algorithms or completely new algorithms for co-processor architectures will play a key role in the future for CPU development as well.

## 1.2. Contributions

In this thesis we take a deep look into using the GPU as co-processor for DBMS operations. We compare the architecture and explain, where the GPU is good at and when

it is better to use the CPU. By doing a series of micro benchmarks we compare the performance of a GPU to typical server CPUs. Against popular opinion the GPU is not necessarily always better for parallel compute-intensive algorithms. Modern server CPUs provide a parallel architecture with up to 8 powerful cores themselves. With the help of the benchmarks we point out a few rules of thumb to identify good candidates for GPU execution.

The reason that the GPU provides so much raw power is that hundreds of cores are fitted onto one board. These cores are simple and orders of magnitude slower than a typical CPU core and because of their simplicity they do not optimize the execution of instructions. Where the CPU uses branch-prediction and out-of-order-execution, on the GPU each instruction specifies what one core does and when. Additionally porting an algorithm to the Single Instruction Multiple Data (SIMD) execution model often involves overhead. Missing optimization and the necessary overhead usually mean that only a fraction of the theoretical power can actually be used. Hence the code itself has to be highly optimized for the task. Because of this we have to differentiate between static tasks that always execute the same code branches in the same order, and dynamic tasks such as query execution, where modular code blocks are chained and executed depending on the query.

For static tasks we show candidates from three different classes: application logic, query optimization, and a maintenance task. While the first two can benefit from the GPU for certain input parameters, the maintenance task is not only data-intensive but also not "compatible" to the GPU's architecture. For dynamic tasks we propose code generation and Just-In-Time (JIT)-compilation to let the compiler optimize the code for the specific use case at run-time. In any case: if the input for a task is too small, not all cores on the GPU can be used and performance is lost. Hence, we have to decide each time depending on the input whether it is worth using the GPU for the job. In the last part of this thesis we present a framework which automatically makes a decision based on predicted run-times for the execution.

We show that

- there is a huge difference between parallel algorithms for the CPU and the GPU.
- application logic can be executed by the DBMS and on the GPU with advantages from both concepts.
- query optimization can benefit from the performance the GPU provides in combination with a library for linear algebra.
- there are tasks that simply cannot benefit from the GPU; either because they are data intensive or because they are just not compatible to the architecture.
- we can execute queries completely on the GPU, but there are limits for what can be done with our approach.
- the system can learn, when using the co-processor for a certain task is beneficial and automatically decide, where to execute a task.

In every evaluation we aim for a fair comparison between algorithms optimized for and running on a modern multi-core CPU against GPU-optimized algorithms. Because DBMS usually run on server hardware, we use server CPUs and GPUs; details are explained in Section 6.1 and in the Appendix.

## 1.3. Outline

In Chapter 2 we start off with explaining the environment for our research: while in disk-based DBMS most tasks were Input/Output (I/O) bound, Main Memory Database System (MMDBMS) make stored data available for the CPU with smaller latency and higher bandwidth. Hence, processing power has a higher impact in these systems and investigating other processing units makes more sense than before.

The most popular co-processor at the moment, the GPU, is explained in detail in Chapter 3. We start with the details about its architecture and the way it can be used by developers. With a series of benchmarks we give a first insight on how a GPU performs in certain categories compared to a CPU. Based on these findings we explain the further methodology of our work in Section 3.5. The terms *static* and *dynamic task* are explained there.

In Chapter 4 we take a deep look at candidates of three different classes of static tasks within a DBMS and explain how these algorithms work and perform on a GPU. While some application logic and query optimization can benefit from the GPU for certain input parameters, the maintenance task is not only data-intensive but also not "compatible" to the GPU's architecture. Even without the transfer bottleneck, the CPU would be faster and more efficient in processing.

However, the most important task of any DBMS is query execution, which we consider to be a dynamic task. Because of the GPU's characteristics we cannot use the approach of executing each database operator on its own as explained in Section 5.2. Instead we use the novel approach of generating code for each individual query and compile it at run-time. We present the approach, its advantages, especially for the GPU, as well as its limitations in Chapter 5.

Most tasks are not simply slower or faster on the GPU. Instead, it depends on the size of the input data and certain parameters that can only be determined at run-time. Because of the unlimited number of different situations and hardware-combinations we cannot decide at compile time, when to use which processing unit. The framework HyPE decides at run-time and is presented in Chapter 6.

# 2. Main Memory Database Management Systems

Relational Database Management Systems (RDBMSs) today have to serve a rich variety of purposes. From simple user-administration and profile storage in a web community forum to complex statistical analysis on costumer and sales data in world-wide acting corporations anything can be found. For a rough orientation the research community categorizes scenarios under two different terms: Online Transactional Processing (OLTP) and Online Analytical Processing (OLAP). Until recently there was no system that could serve both workloads efficiently. If a user wanted to do both types of queries on the same data, replication from one system to another was needed. Nowadays, the target of the major DBMS vendors is to tackle both types efficiently in one system. The bottleneck of I/O operations in traditional disk-based DBMS makes it impossible to achieve this goal.

The concept of the MMDBMS eliminates this bottleneck and gives the computational power of the processor a new higher priority. Because I/O operations are cheaper on Random Access Memory (RAM), processes that were I/O-bound in disk-based system are often CPU-bound in MMDBMS. Co-processor provide immense processing speeds for problems of their domain. Their integration into a MMDBMS is a new chance to gain better performance for such CPU-bound tasks.

In this chapter, we explain how MMDBMSs work and what possibilities and challenges arise because of their architecture. In the first section we explain the difference between OLTP and OLAP workloads, which is necessary for a basic understanding of the challenges DBMS researchers and vendors face. The following Section 2.2 explains the recent changes in hardware that explain the rising interest in MMDBMSs. We take a look at the architecture and techniques used in HANA to support both scenarios, so-called mixed workloads.

For this thesis we focus on the GPU as co-processor, but there are of course other candidates. Before we get to the details of GPUs in the next chapter, we take a look at noticeable attempts that were already made to integrate different types of co-processors into a DBMS in Section 2.3.

## 2.1. OLTP and OLAP

Based on characteristics and requirements for a DBMS there are two fundamentally different workloads: OLTP and OLAP.

A DBMS designed for OLTP must be able to achieve a high throughput of read and—at the same time—write transactions. It must be able to keep the data in a consistent

state at every time and make sure that no data is lost even in case of a power failure or any other interruption. Additionally, transactions do not interfere in each others processing and are only visible, when they are completed successfully. These characteristics are know under the acronym ACID: Atomicity, Consistency, Isolation and Durability [41]. Users and applications that used RDBMSs rely on the systen to act according to these rules. Typical Structured Query Language (SQL) statements of a OLTP workload are shown in Listing 2.1. They are a mixture of accessing data with simple filter criteria, updating existing tuples and the insertion of new records into the system.

Listing 2.1: Typical OLTP queries

```
1 SELECT * FROM customer
2   WHERE customer_id=12;
3
4 INSERT INTO customer(name, address, city, zip)
5   VALUES("Cust Inc", "Pay Road 5", "Billtown", 555);
6
7 UPDATE customer SET customer_status="gold"
8   WHERE name="Max Mustermann";
```

The requirements for a OLAP-optimized DBMS are different. Instead of simple and short transactions, they must support the fast execution of queries that analyze huge amounts of data. But in contrast to OLTP scenarios the stored data is not or only rarely changed. Therefore, the ACID rules do not play an important role in analytic scenarios. Compared to a OLTP scenario there are less queries, but every query accesses large amounts of data and usually involves much more calculations due to its complexity. A typical OLAP-query is shown in Listing 2.2.

Listing 2.2: Query 6 of the TPC-H benchmark [100]

```
1 SELECT sum(l_extendedprice * l_discount) as revenue
2 FROM    lineitem
3 WHERE l_shipdate >= date ':1'
4         AND l_shipdate < date ':1' + interval '1' year
5         AND l_discount BETWEEN :2 - 0.01 AND :2 + 0.01
6         AND l_quantity < :3;
```

Although classic RDBMSs support OLTP and OLAP workloads, they are often not capable of providing an acceptable performance for analytical queries. Especially in enterprise scenarios another system is used that is optimized solely for OLAP. These so-called *data warehouses* use read-optimized data structures to provide fast access to the stored records. Furthermore, they are optimized to do calculations on/with the data while scanning it. Because the data structures used do not support fast changes, OLAP systems perform poorly at update- or write-intensive workloads. Therefore, the data that was recorded by the OLTP system(s) must be transferred regularly. This happens in form of batch inserts into the data warehouse. Since this process also involves some form

of transformation to read-optimized schemes and data structures, it is called Extract, Transform, Load (ETL) [102]. The batch insert can contain data from a high number of different system. Their exports are converted and may be pre-processed, e.g., sorted and aggregated, before they are actually imported into the OLAP system [58].

A typical business example that shows the organization of different DBMS, is a global supermarket chain. Every product that is sold has to be recorded. However, the transaction is not completed when the product is registered by the scanner at checkout, but when the receipt is printed. Until then it must be possible to cancel the transaction without any change to the inventory or the revenue. Additionally, in every store hundreds or thousands transaction are made every day. This is a typical OLTP workload for a DBMS. Similar scenarios can be found at a warehouse, where the goods are distributed to the stores in the region. Here, every incoming and outgoing product is recorded just in time. In contrast to that a typical OLAP-query is to find the store with the lowest profit of a certain region, or the product that creates the highest revenue in a month. This OLAP workload is usually handled by the data warehouse, which holds the records of all stores and warehouses.

This solution has disadvantages: since at least two DBMS are involved, there is no guarantee that data is consistent between both. During the ETL process, it must be ensured that no records are lost or duplicated. Additionally, because there is not one single source of truth, every query has to be executed in the right system. But the most significant disadvantage is that the more sources there are, the longer the ETL process takes. Hence, the time until the data is available for analysis gets longer, when more data is recorded. Especially in global companies *real-time analytics*[1] are not available with this approach [83].

## 2.2. Main Memory DBMS for a Mixed Workload

Most systems today use Hard Disk Drives (HDDs) as primary storage, because they provide huge amounts of memory at an affordable price. However, while CPU performance increased exponentially over the last years, HDD latency decreased and bandwidth increased only slowly. Table 2.1 lists the access latency of every memory type available in modern server hardware. There is a huge difference between HDD and RAM latency (and bandwidth as well), which is known as the *memory gap*. Therefore, the main performance bottleneck for DBMS nowadays (and at least in the last 20 years) is I/O. The first DBMSs that addressed this problem by holding all data in main memory while processing queries were available in the 1980s [26]. These MMDBMSs do not write intermediate results, e.g., joined tables, to disk like traditional disk-based system and therefore reduce I/O, especially for analytic queries, where intermediate results can become very large.

In recent years the amount of RAM fitting into a single machine passed the 1 TB mark and became cheap enough to replace the HDD in most business scenarios. Hence, the

[1] This has of course nothing to do with the definition of *real-time* in computer science.

| | |
|---|---|
| L1 Cache | ≈4 cycles |
| L2 Cache | ≈10 cycles |
| L3 Cache | ≈40–75 cycles |
| L3 Cache via Intel QuickPath Interconnect (QPI) | ≈100–300 cycles |
| RAM | ≈60 ns |
| RAM via QPI | ≈100 ns |
| Solid State Drive (SSD) | ≈80 000 ns |
| HDD | ≈5 000 000 ns |

Table 2.1.: Approximation of latency for memory access (reads) [64, 30, 105]

concept of MMDBMS was extended to not only keep active data in main memory, but use RAM as primary storage for all data in the system[2].

For DBMS vendors this poses a new challenge, because moving the primary storage to RAM means tuning data structures and algorithms for main memory characteristics and the second memory gap between CPU-caches and RAM. MMDBMS still need a persistent storage to prevent data loss in case of a power failure, therefore every transaction still triggers a write-to-disk operation. Hence, OLTP workloads can by principle not benefit as much as OLAP, but modern hardware enables these systems to provide acceptable performance for OLTP as well.

In the next sections we show some of the principle design decisions of relational MMDBMS on the example of SAP HANA. While other commercial RDBMS started of as disc-oriented systems and added main memory as primary storage in their products—e.g., SQLServer with Hekaton [20], DB2 with BLU [86], and Oracle with Times Ten [80]—HANA is built from the ground up to use main memory as the only primary storage.

### 2.2.1. SAP HANA Architecture

On the one hand SAP HANA is a standard RDBMS that supports pure SQL and, for better performance, keeps all data—intermediate results as well as stored data—in main memory. Hence, the system provides full transactional behavior while being optimized for analytical workloads. The design supports parallelization ranging from thread and core level up to distributed setups over multiple machines. On the other hand HANA is a platform for data processing specialized to the needs of SAP applications [21, 92, 22].

Figure 2.1 shows the general SAP HANA architecture. The core of the DBMS are the In-Memory Processing engines, where relational data can either be stored row-wise or column-wise. Data is usually stored in the column store, which is faster for OLAP scenarios as we will explain in Section 2.2.2. However, in case of clear OLTP workloads, HANA also provides a row store. The user of the system has to decide, where a table is stored at creation time, while the system handles the transfer between the storage engines when a query needs data from both. Hence, the ETL-process is not necessary,

[2]To emphasize the difference SAP HANA is sometimes called In-Memory DBMS

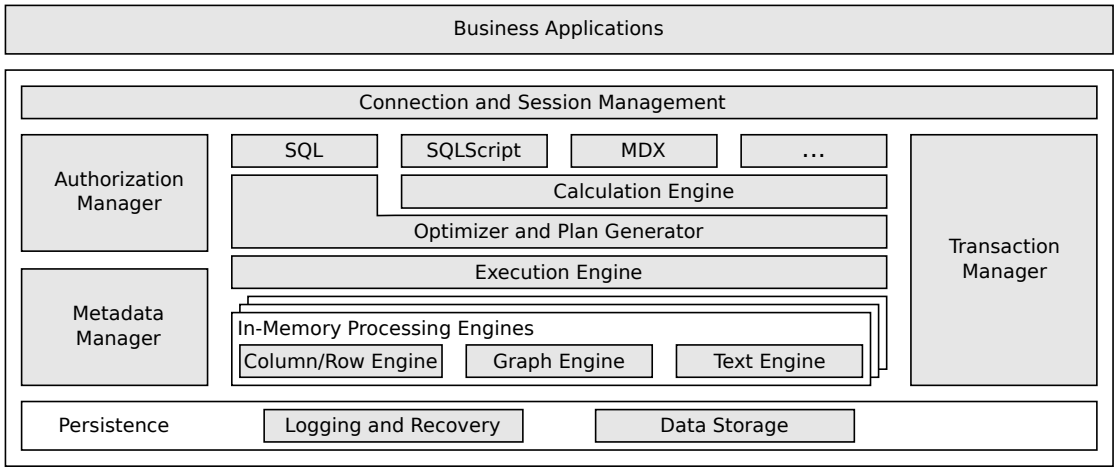



Figure 2.1.: Overview of the SAP HANA DB architecture [22]

resp. transparent to the users of the DBMS. Psaroudakis et al. showed how HANA handles both types of workloads at the same time [85].

As a data management platform, HANA allows graph data and text data to be stored in two specialized engines next to relational data. In every case, processing and storage are focused on main memory. If necessary tables can be unloaded to disk, but have to be copied to main memory again for processing. Because of this there is no need to optimized data structures in any way for fast disk access. Instead data structures as well as algorithms are built to be cache-aware to cope with the second memory gap. Furthermore, the engines compress the data using a variety of compression schemes. As we discuss in Section 2.2.3 this not only allows more data to be kept in main memory but speeds up query execution as well.

Applications communicate with the DBMS with the help of various interfaces. Standard applications can use the SQL interface for generic data management functionality. Additionally, SQLScript [9] enables procedural aspects for data processing within the DBMS. More specialized languages for problems of a certain domain are MDX for processing data in OLAP cubes, WIPE to query graphs [90, 89], and even an R integration for statistics is available [39]. SQL queries are translated into an execution plan by the plan generator, which is then optimized and executed. Queries from other interfaces are eventually transformed into the same type of execution plan and executed in the same engine, but are first described by a more expressive abstract data flow model in the calculation engine. Independent of the interface, the execution engine handles the parallel execution and the distribution over several nodes.

On top of the interfaces there is the Component for Connection and Session Management, which is responsible for controlling individual connections between the database layer and the application layer. The authorization manager governs the user's permissions and the transaction manager implements snapshot isolation or weaker isolation levels. The metadata manager holds the information about where tables are stored in which form and on which machine if the system was set up in a distributed landscape. All data is kept in main memory, but to guarantee durability in case of an (unexpected)

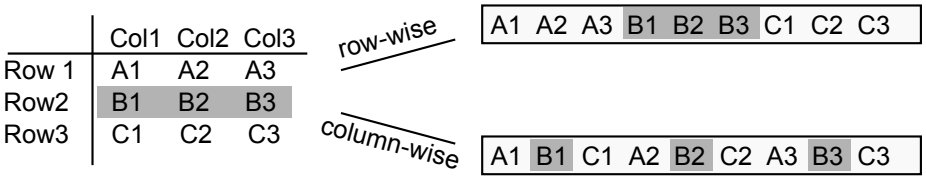



Figure 2.2.: Row- vs. column-oriented storage

shutdown data needs to be stored on disk (HDD or SSD) as well. During savepoints and merge operations (see Section 2.2.4) tables are completely written to disk; in between updates are logged. In case of a system failure, these logs are replayed to recover the last committed state [22].

### 2.2.2. Columnar Storage

Since relations are two-dimensional structures there has to be a form of mapping to store them in the one-dimensional address space of memory. In general there are two different forms: either relations are stored row-wise or column-wise.

As shown in Figure 2.2 in row-wise storage all values of a tuple are stored contiguously in memory. The most significant advantage is that the read access to a complete row as well the insertion of a new row, which often happens in OLTP scenarios, can be processed efficiently. In contrast column stores store all values of a column contiguously in memory. Therefore, access to a single row means collecting data from different positions in memory. In OLAP scenarios values of single attributes are regularly aggregated, which means that fast access to the whole column at once is beneficial. When queries access only a small number of columns in the table columnar storage allows to just ignore the other columns. Naturally, column stores have the advantage in these situations.

Main memory is still an order of magnitude more expensive than disk. Therefore, MMDBMSs use compression whenever possible before storing the data. In terms of compression, the columnar storage has an advantage: the information entropy of data stored within one column is lower than within one row. Every value is by definition of the same data type and similar to other values. Lower information entropy directly leads to better compression ratios [2]. It is easier, for instance, to compress a set of phone numbers than a set with a phone number, a name and an e-mail address. Additionally, often there is a default value that occurs in columns at a high frequency.

There are two different data layouts for column stores. MonetDB for instance uses the Decomposed Storage Model (DCM), which was first described by Copeland and Khoshafian [16]. Every value stored in the column gets an identifier to connect it with the other values of the same row, i.e., the row number of every value is explicitly stored. In MonetDB's Binary Association Table (BAT) structure, there are special compression schemes to minimize the overhead of storing this identifier, e.g., by just storing beginning and end of the range.

The second layout—which is used by HANA—stores every column of a table in the

same order, so that the position of a value inside the column automatically becomes the row identifier. While the overhead for the row identifier is not there in this layout, individual columns cannot be re-ordered, which often is advantageous for compression algorithms (cf. next Section).

### 2.2.3. Compression

Compression in DBMSs is a major research topic. Not only does it save memory, it makes it also possible to gain performance by using compression algorithms that favor decompression performance over compression ratio [109, 1]. The reason is that, while we had an exponential rise in processing power for decades, memory bandwidth increased only slowly and latency for memory access, especially on disk, but also for main memory has been almost constant for years [83]. The exponential growth in processing power was possible because transistors became constantly smaller. Therefore the number of transistors per chip doubled every 18 months. This effect was predicted in the 1960s [69] and became know as Moore's law. While the number of transistors directly influences processing power and memory capacity, it does not necessarily affect memory latency and bandwidth, neither for disk nor for main memory. Hence, Moore's law cannot be applied to these characteristics. There was some growth in bandwidth (for sequential access!) but especially seek times for hard disks have not changed in years because they are limited by mechanical components. Main memory development was also slower than the CPU's.

Therefore, even in MMDBMS I/O to main memory is the bottleneck in many cases, e.g., scanning a column while applying filter predicates is memory bound even with compression [103]. Modern multi-core CPUs are just faster at processing the data than reading it from memory. By using compression algorithms we do not only keep the memory footprint small, we also accelerate query performance. This is of course only true for read-only queries, not for updates. Updates on compressed structures are a major problem, because they involve decompression, updating de-compressed data and re-compression of the structure. To prevent this process happening every time a value is update, they are usually buffered and applied in batches. We describe the details of this process in Section 2.2.4.

In HANA's column store different compression schemes are used on top of each other. We describe the most important ones in the following paragraphs.

#### Dictionary Encoding

A very common approach to achieve compression in column stores is dictionary or domain encoding. Every column consists of two data structures as depicted in Figure 2.3: a dictionary consisting of every value that occurs at least once in the column and an *index vector*. The index vector represents the column but stores references to the dictionary entries, so called *value IDs*, instead of the actual values.

In many cases the number of distinct values in one column is small compared to the number of entries in this column. In these cases the compression ratio can be immense.

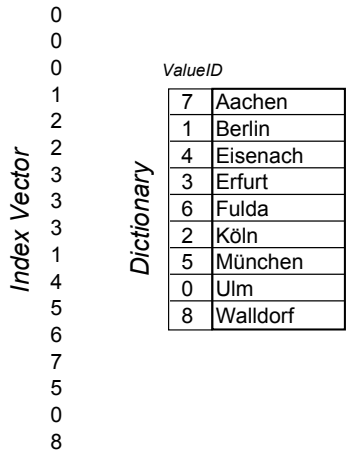



Figure 2.3.: Dictionary compression on the example of a city column

For typical comment columns or alike, in which almost every value is unique, there is mostly no effect on the memory footprint. If every value is distinct, even more space is needed.

Although numeric columns do not benefit from this type of compression in terms of memory usage, HANA stores all regular columns in dictionary encoded form. There are various reasons for that, e.g., the execution of some query types can be done more efficiently with the help of the dictionary. Obviously `distinct` queries can be answered by just returning the dictionary. Also, query plans for SQL-queries with a `group by` clause for example can be optimized better because the maximum number of rows in the result can be easily calculated because the size of the dictionary is known. But the most important reason for dictionary encoding is that compression techniques for the index vector can be applied independently of the type of the column.

### Bit-Compression for Index Vectors

In combination with dictionary encoding the most efficient compression algorithm for the index vector is bit-compression. Usually the value IDs would be stored in a native, fixed-width type such as *integer* (or *unsigned integer*), i.e., every value ID needs 32 bit. Since we know the maximum value ID $v_{max}$ that occurs in the index vector—the size of the dictionary—we also know that all value IDs fit into $n = \lceil \log_2 v_{max} \rceil$ bit. Hence, we can just leave out the leading zeros, which makes bit-compression a special form of prefix-compression. With this approach we do not only achieve a high compression ratio in most scenarios, but we also gain scan performance. Because we are memory bound while scanning an uncompressed column and the de-compression of bit-compressed data can be implemented very efficiently with SIMD-instructions, the scan throughput is higher with compression [104, 103] than without. Moreover, bit-compression allows the evaluation of most predicates, e.g., the `where`-clause of SQL-statements, on-the-fly without de-compressing the value IDs. The performance depends strongly on the bit-

case, i.e., the size of one value ID in memory, but even with complex predicates we are still memory-bound when scanning with one thread only [103].

In contrast to the other compression techniques shown here, bit-compressed data can never be greater than the uncompressed value IDs. The only draw-back of this form of compression is that the direct access to a single value requires the decompression of one block of values. In most cases, however, the initial cache-miss dominates the access time. Therefore, similar to dictionary encoding this type of compression is used for every stored index vector in HANA.

#### Further Light-Weight Compression (LWC) Techniques

Additionally there are other forms of compression techniques used within HANA. One of the major requirements is that compressed data can be de-compressed fast while scanning and also while accessing single values, i.e., they are light-weight. Sophisticated algorithms, such as bzip2 or LZMA usually achieve a better compression ratio, but are just too slow for query execution. While most of the LWC-techniques work well for sequential scanning of a column, access times for single values vary.

In Figure 2.4 the used techniques for compression in HANA to date are depicted. They are explained in detail in [63].

*Prefix coding* eliminates the first values of the index vector when they are repeated. In most columns there is no great benefit to use it, except for sorted columns. However, this compression type introduces almost no overhead and does not affect scan performance. Therefore, it is often combined with other techniques.

*Sparse coding* works by removing the most frequent value IDs from the index vector and managing an additional bit vector, where the positions of removed value IDs are marked. Especially the default value often occurs in columns, therefore this algorithm achieves a good compression ratio in most cases. A disadvantage is that direct access requires the calculation of the value ID's position in the compressed index vector by building the prefix sum of the bit vector. Depending on the position of the requested value ID this leads to a large overhead compared to the simple access.

*Run-length encoding* (RLE) (slightly modified version of [32]) compresses sequences of repeating value IDs by only storing the first value of every sequence in the index vector and the start of each sequence in another vector, which can also be bit-compressed. While this may achieve good compression ratios for columns, where values are clustered, there is the possibility that the compressed structure may need even more memory than the uncompressed index vector. Single access to a value ID at a given position requires a binary search in the vector that holds the starting positions and can therefore be expensive.

For *cluster coding* the index vector is logically split into clusters with a fixed number of value IDs. If one partition holds only equal value IDs, it is compressed to a single value ID. A bit vector stores whether the partition is compressed or not. Similar to RLE this achieves good ratios for clustered values, but the compression is limited by the chosen cluster size. Like with sparse coding a single lookup requires the calculation of the prefix sum for the bit vector and can therefore be expensive. Cluster coding can

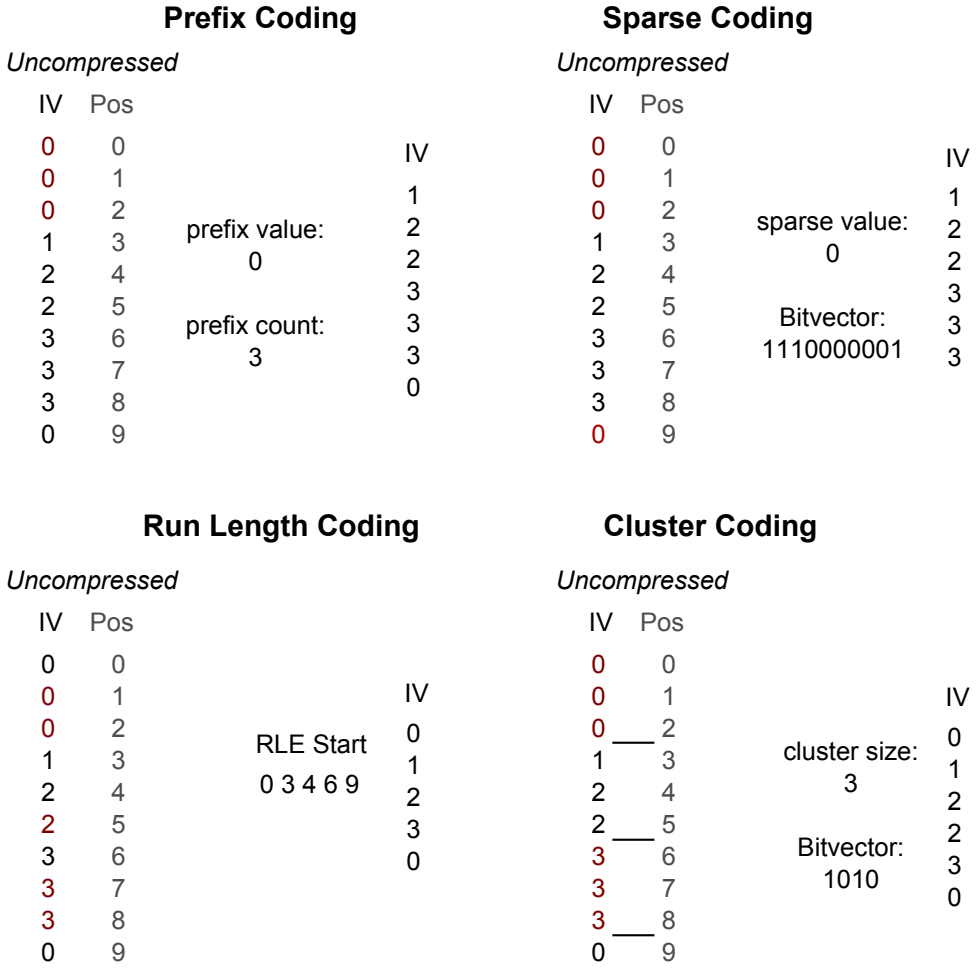



Figure 2.4.: Compression algorithms for the index vector

be extended to *indirect coding* by creating a dictionary for a cluster that contains more than one value ID.

Except for prefix compression the direct access to the value ID at a given position gets more expensive the larger the index vector gets. To limit the time needed for such an access, every compression is applied to blocks of values with a fixed size. This way every single lookup requires to calculate the right block number and the access to the structure that stores the compression technique for this block. Afterwards, the block needs to be de-compressed to access the value. The block-wise approach also has the advantage that the best compression algorithm can be chosen for parts of the index vector. Overall this gives a better compression ratio than using one technique for the complete column [63].

#### Re-Ordering Columns for Better Compression Ratios

All compression techniques shown in the last section do work best for columns that are sorted by the frequency of their value IDs. Especially having the most frequent value ID on top is the best situation in every case, because we can combine all techniques with prefix-coding (of course this is not necessary for RLE). Fortunately, relations do not guarantee any order. Therefore, HANA is free to re-arrange the rows to gain better performance or a smaller memory footprint. Of course, the columns cannot be sorted independently since this would require to map the row number to the position of a value ID. In most cases this would be more expensive (in terms of memory footprint and compression) than having no LWC at all. Hence, the system has to determine which column it sorts for a good performance. Finding the optimal solution for this problem is NP-complete [5]. Lemke et al [63] propose greedy heuristics to find a near-optimal solution. The key to finding a good solution is a combination of only considering columns with small dictionaries and applying the compression techniques to samples of each of these columns.

#### Compression of String Dictionaries

In typical business scenarios up to 50% of the total databases size are the dictionaries of string columns [87]. In many cases the largest of these dictionaries are the ones that are accesses rarely. In the TPC-H benchmark the *comment* column of the *lineitem* table is a typical example [100]. While it is the largest of all columns—uncompressed it is one fourth of the whole database size—there is no query that accesses it. The *comment* column of the *orders* table, one eighth of the total, is accessed in one of the 22 queries of the benchmark. The compression of these dictionaries is therefore essential to bring the memory footprint down, while there is no impact on the performance in many cases. Hence, in contrast to index vectors, it can also make sense to use heavy-weight compression on some string dictionaries. But the candidates have to be carefully chosen, since access to certain dictionaries may be performance critical. Ratsch [87] evaluates known compression schemes for string dictionaries in HANA and compares there suitability for common benchmarks. He achieves compression rates between 30% and 50% without a significant impact on the scan-performance. However, the time

needed for initially compressing the dictionaries has to be considered as well. The work shows that automatically determining the right scheme is a complicated task and cannot be done without access statistics and sample queries on the compressed data.

### 2.2.4. Main and Delta Storage

Although the compression techniques presented in the last section have different characteristics in terms of compression ratios, scan-, and single-lookup-performance, they have one thing in common: they have a noticeable impact on the write-performance. In many cases the update of a single value triggers the de- and re-compression of the whole structure that holds the column. The insertion of one row into a large table might take seconds, especially with all the algorithms that are executed to evaluate the optimal compression scheme for parts of the table. This therefore contradicts the original goal of building HANA as a MMDBMS to support mixed-workloads.

Since column stores have a poor write performance in general because of the memory layout, this is a common problem. The general idea is to buffer updates in a secondary data structure, while keeping the main storage static. C-Store [98], for instance, differentiates between *Read-optimized Store (RS)* and *Writeable Store (WS)* [94]. A so called *Tuple Mover* transfers data in batches from WS to RS.

HANA's update buffer, called *Delta Storage*, is optimized to keep read- and write-performance in balance. It stores data in columns just like the main storage, but because it usually holds less data than the main storage, none of the compression techniques, except for dictionary encoding, are used. The delta dictionary is independent of the main dictionary, i.e., values may appear in both dictionaries with different value IDs. In consequence a insertion operation only triggers a lookup in the delta dictionary. In case the value is already there, the value ID is inserted into the column. If the value is not in the column, it is inserted and a new value ID is assigned. While the main dictionary is stored as a sorted array, where the value ID is the position of a value inside the array, the delta dictionary has to use a structure which consists of two parts to provide fast look-ups and insertions. First, values are stored in an array in order of their insertions. The value ID is the position of the of the value within the array. However, finding the value ID belonging to a value takes $\mathcal{O}(n)$ operations. Therefore, values are also inserted in a B+-Tree, where they are used as key and the value ID is the (tree-)value. This way, the value ID belonging to a value can be found with $\mathcal{O}(\log n)$ operations. Insertions have the same complexity.

HANA uses an insert-only approach, i.e., update operations delete the old entry and insert a new one. Because removing the row from main storage immediately would be expensive, it is marked invalid in a special bit-vector instead and finally removed when the delta is merged into the main storage. HANA also supports temporal data, then the bit-vector is replace with two time stamps that mark the time range for which the entry was valid [55].

With the delta approach the main storage can be optimized solely for reading and it can be stored on HDDs as is to guarantee durability. In contrast the delta storage is kept only in main memory, but operations are logged to disk. In case of a a system

failure, the main storage can simply be loaded and the delta log has to be replayed to return to the last commited state. Of course data has to be moved from delta buffer to main storage at some point in time. This process—called Delta Merge—creates a new main dictionary with entries from the old main and delta dictionary. Afterwards all value IDs have to be adjusted to the new dictionary. This can be done without look-ups in the new dictionary by creating a mapping from old to new IDs while merging the dictionaries. During the whole operation all write transactions are redirected to a new delta buffer, read transaction have to access the old main and delta as well as the new delta storage. Except for the short time, when the new delta and main are activated, there is no locking required. In Section 4.3 we take a closer look at the creation of the new dictionary.

## 2.3. Related Work: Using Co-processors in a DBMS

Before we focus on GPUs for the remainder of this thesis, we take a look at related work that concentrates on other available co-processors in the context of DBMS.

FPGAs are versatile vector processors that provide a high throughput. Müller and Teubner provide an overview of how and where FPGAs can be used to speed up DBMS in [70]. One of there core messages is that an FPGA can speed up certain operations but should—like the GPU—be used as a co-processor next to the CPU. In [97] the same authors present a concrete use case: doing a stream join on the FPGA.

The stream join was also evaluated on the cell processor in [29, 28]. However, the cell processor is not a typical co-processor but a heterogeneous platform that provides different types of processing units, the PowerPC Processing Element (PPE) and 8 Synergistic Processing Elements (SPEs). The PPE is similar to a typical CPU core, while the SPEs work together like a vector machine. The co-processing concept is inherent in this architecture, because the developer has to decide which processing unit to use for every algorithm. We will face similar challenges in the future with heterogeneous architectures that provide CPU and GPU on the same chip. The same authors also published work about sorting on the cell processor [27].

Recently Intel announced its own co-processor: the Xeon Phi, also called Many Integrated Core (MIC), which is a decendant of Larrabee [91]. For the Xeon Phi, Intel adapted an older processor architecture and inter-connected a few dozen cores with a fast memory bus. The architecture focuses on parallel thread execution and SIMD instructions with wider registers than Xeon processors. It still supports x86 instructions, so a lot of software can be ported by re-compiling it. However, to gain performance algorithms have to be adjusted or re-written to support a higher number of threads and vector-processing. Intel published a few insights on the architecture on the use case of sorting in [71]. In 2013 the Xeon Phi also started supporting OpenCL. Therefore, it should be possible to use algorithms written for the GPU. Teodoro et al. compared the performance of GPU, CPU and Xeon Phi in [96].

In a typical computer system there are other co-processors next to the GPU that can be used for general purpose calculations. Gold et al. showed that even a network

processor can be used to execute hash joins [31].

# 3. GPUs as Co-Processors

In the last decade GPUs became more and more popular for general purpose computations that are not only related to graphic's processing. While there are lots of tasks from different fields ported to the GPU [82], the best known and maybe most successful one is password cracking. With the help of commodity hardware in form of graphic cards it became possible for almost everyone to guess passwords in seconds instead of hours or even days [51]. This shows that the GPU has an immense potential to speed up computations. Additionally, it seems that the GPU will be integrated directly into almost every commercial CPU in the future. Therefore, we think that it is the most important co-processor architecture at the moment and concentrate on GPUs for general purpose calculations for the remainder of the thesis.

GPUs use a different architecture than CPUs to process data in a parallel manner. There are similarities with other co-processors, such as FPGAs and Intel's recently released architecture Xeon Phi. While CPUs use a low number of powerful cores with features such as branch prediction and out-of-order-processing, FPGAs, GPUs, and Xeon Phis consist of dozens to hundreds of weak processing units, which have to be used in parallel to achieve good (or even acceptable) performance. Therefore most of the findings in this thesis also apply to other co-processors, especially if they can be programmed with the OpenCL framework.

We describe the key aspects of the GPU's architecture and compare it to the CPU in Section 3.1. The main problem when offloading calculations to an external co-processor is, that it has to communicate with the CPU to get input data and transfer the results back. We discuss the influence of this bottleneck in Section 3.2. To interact with GPUs developers have the choice between direct programming and the usage of libraries that transparently process data on the GPU. OpenCL[1] and CUDA[2] are the two most common frameworks to program GPUs directly. There characteristics and available libraries for the GPU are described in Section 3.3. The main part of this chapter in Section 3.4 focuses on different micro benchmarks to show the strong- and weak-points of the GPU. Based on the characteristics we can derive from these benchmarks we draw a conclusion and outline the remainder of the thesis in Section 3.5.

## 3.1. SIMD Processing and Beyond

Flynn specified four classes of machine operations [24], which are still used today to differentiate between parallel processor architectures. We introduce the taxonomy in

[1] http://www.khronos.org/opencl/
[2] http://www.nvidia.com/cuda

Subsection 3.1.1. However, neither CPU nor GPU fall into only one category. Both architectures are mixed, but have a tendency. In Section 3.1.2 we discuss this further.

### 3.1.1. Flynn's Taxonomy

The four classes of Flynn's Taxonomy are shown in Figure 3.1:

**Single Instruction Single Data (SISD)** describes a processor with one processing unit that works on one data stream. In this work we often refer to this as sequential processing. Before the first SIMD instruction sets, e.g., MMX, 3DNow!, and Streaming SIMD Extensions (SSE), were introduced, single core processors were working in a pure SISD manner.

**Single Instruction, Multiple Data (SIMD)** means that a number of processing units are executing the same statements on different data streams. In the 1970s vector processors were working like that. Today most processors (CPUs and GPUs) have parts that work in a SIMD fashion.

**Multiple Instruction, Single Data (MISD)** requires a number of processing units to do different instructions on the same data stream. This processing model can be used for fault tolerance in critical system. Three or more processors do the same operation and there results are compared.

**Multiple Instruction, Multiple Data (MIMD)** describes processing units that can independently work on different data streams. In contrast to just using multiple SISD processors this approach allows to share resources that are not needed exclusively, e.g., the memory bus or caches.

Multi-Core CPUs can be roughly classified as MIMD and GPUs as SIMD architectures. However, both have aspects of the other architecture. The MIMD-nature of modern multi-core CPUs is obvious to most developers. Threads can work independently on different algorithms and data segments in at the same time. However, with instruction sets such as SSE and Advanced Vector Extensions (AVX) it is also possible to parallelize execution within one thread, e.g., building the sum of multiple integers with one instruction. This SIMD aspect of the CPU is almost hidden, because in most cases this type of processing happens without the developer's explicit interference. The compiler analyses the code and uses SIMD-instructions automatically. Nevertheless these instructions can be used explicitly and the benefit may be dramatic in some cases [104, 103].

Nowadays, no (pure) SISD processors can be found in server, desktop, or mobile computers. Even single-core CPUs usually have more than one processing unit to execute SIMD instructions. Multi-Core CPUs and GPUs, as well as Xeon Phis, are a mixture of SIMD and MIMD processors. However, the SIMD aspect of GPUs is important because of the high number of cores that work in single lock step mode. On the CPU SIMD is available in form of instructions, which are used by modern compilers automatically.

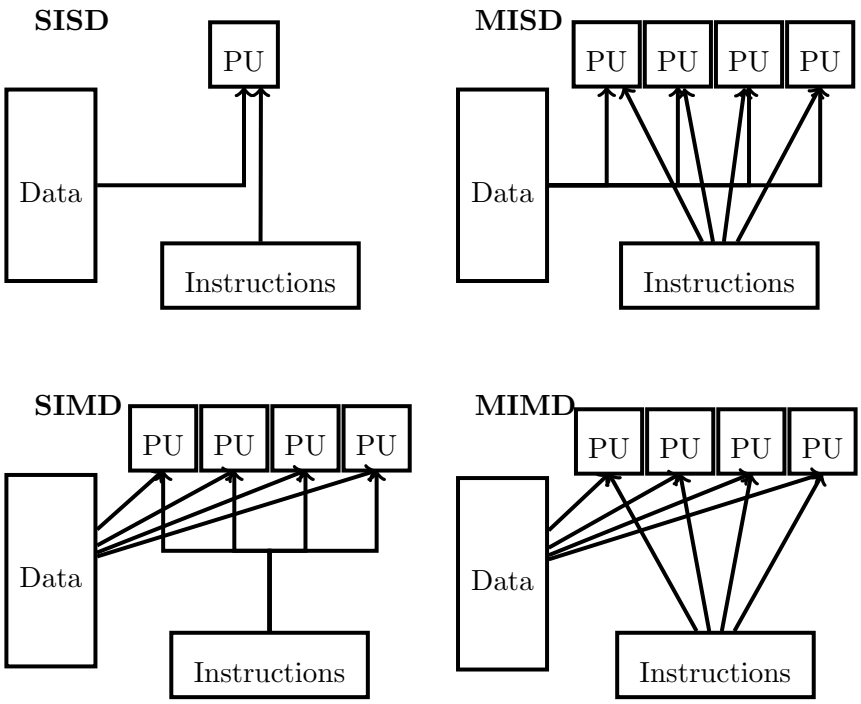



Figure 3.1.: Flynn's taxonomy

### 3.1.2. Hierarchy of Parallelism—the GPU Architecture

While GPUs are often referred to as SIMD processors, its architecture is actually MIMD-SIMD-hybrid. Figure 3.2 depicts the simplified *NVIDIA Kepler GK110* architecture, which we explain exemplary in this section. We use the NVIDIA terminology for the description, the concepts in principle are similar to hardware of other vendors.

The 2880 processing cores on the GPU are grouped in 15 Streaming Multi Processors (SMXs). 192 cores on each SMX share a single instruction cache, 4 schedulers, and a large register file. They are interconnected with a fast network, which among other thins provides (limited) access to registers of other cores. Hence, all the cores of one SMX are tightly coupled and designed for SIMD processing. Consequently, threads on the cores are executed in groups of 32. Such a group is called a *warp* and is able to execute only 2 different instructions per cycle, i.e., in every case at least 16 cores execute exactly the same instruction, they run in single-lock-step mode.

Compared to the internals of a SMX the coupling between different SMXs of a GPU is loose. Threads are scheduled in blocks by the so-called GigaThread engine, there is no global instruction cache, and the cores of different SMXs cannot communicate directly. In consequence, no synchronization between threads of different SMXs is possible. They operate independently and represent the MIMD processing part of the GPU's architecture. Hence, one SMX works similar to a vector machine, but the whole GPU is more like a multi-core vector machine.

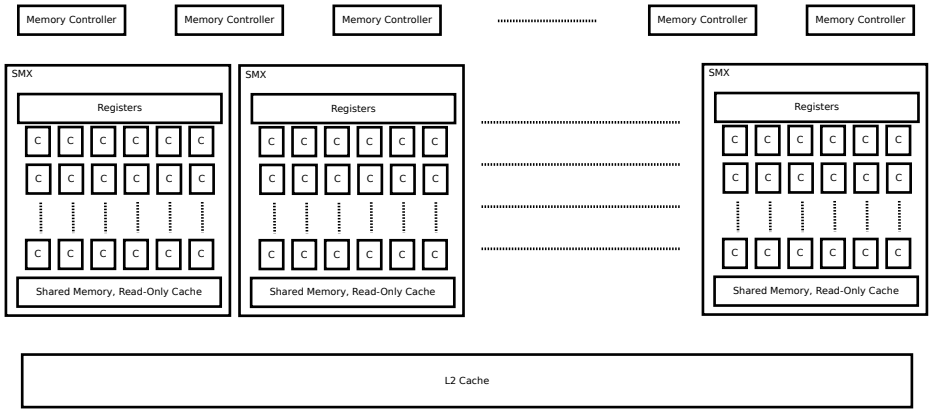



Figure 3.2.: Simplified architecture of a GPU

The memory hierarchy supports this hybrid processing model. Not only do processing cores of one SMX share a register file, they also have fast access to 64 kB of *Shared Memory*. This can be configured to either act as L1 cache or as memory for fast data exchange. In each SMX there is additional read-only and texture memory, which is optimized for image processing. Between SMXs there is no other way to share data than using DRAM. While DRAM on the GPU usually has a much higher bandwidth than the RAM used for CPUs it also has a much higher latency (a few hundred cycles in modern GPUs). In contrast to CPU processing, where thread/context-switches wipe registers, on the GPU every threads keeps its own registers in the register file. Therefore, it is possible to load one warp's data from DRAM to the registers while other warps are running. This method of overlapping is the key for good performance on the GPU, but makes it necessary to execute much more threads than cores. Otherwise, most threads wait for DRAM, while only a few can be executed. Depending on the scenario this under-utilization happens when there are less than be 10 to 100 threads per core. Modern GPUs address the problem of the DRAM's high latency by providing one L2 cache for all SMXs that operates automatically—similar to the L2 cache on CPUs. On Kepler it has a size of 1538 kB. Since memory is always accessed by a half-warp, the GPU is optimized for *coalesced* access. Agian this is simplified but it means that consecutive threads should access consecutive memory addresses. More information about this topic and details about Kepler can be found in the best practices guide for CUDA [73] and the architecture whitepaper [77].

### 3.1.3. Programming Model for GPU Parallelsim

The hybrid architecture of multiple SIMD-processors requires a different way of applying parallel aspects to algorithms. With threads as parallel execution paths on the one hand the differences in SIMD and MIMD cannot be expressed. Simple parallel instruction and paradigms such as OpenMP's *parallel for* on the other hand are not flexible enough; the same counts for special SIMD instructions such as Intel's SSE or AVX. Therefore,

a model of workgroups and threads to differentiate between the MIMD and the SIMD aspects was introduced. Both, OpenCL and CUDA are implementations of this model, we take a look at their differences in Section 3.3.

When you launch a function on the GPU, it will start a fixed number of workgroups and each workgroup consists of a fixed number of threads. Although this is not exact, it makes sense to imagine that one SMX executes a workgroup with all its cores. Workgroups run independent of each other and can only be synchronized by ending the function and starting a new one. In contrast to the workgroups themselves their threads can be easily synchronized and are able to exchange data via a small block of shared memory, which can be accessed with a very low latency (a few cycles depending on the hardware). The number of threads per workgroup is usually called workgroup size. Since at least one warp is always executed, it makes sense to choose a multiple of 32 as workgroup size. In some situations it is useful to know that a group of 32 threads needs no explicit synchronization, but in most cases hardware and compiler take care of an optimization like this. Hence, threads of one workgroup should be handled as if they run in lock-step mode, i.e., every thread executes the same instruction on data in its registers in one cylce.

## 3.2. Communication between CPU and GPU

The way of communication between the CPU and the GPU depends on the GPU's type. In general, we can differentiate between three classes of co-processors by looking at their integration into the computer's CPU.

1. Co-processors such as most graphic cards today are connected via the PCI-Express (PCIe) bus, i.e., data has to be transferred to the graphic cards memory, where the GPU can access it. For data-intensive tasks this is the bottleneck [37]. Intel's XeonPhi and FPGAs are also usually connected via PCIe.

2. The second class of co-processors is tightly integrated into the system bus. The GPU part of AMD's APUs fits into that category. It still operates on its own but it shares memory with the CPU [17]. Therefore data can be transferred much faster.

3. Modern CPUs consist of a collection of co-processors of the third class. The FPU for instance is now integrated and has special registers on the CPU. Cryptographic processors also fall into this category. There are special instructions for these co-processors in the CPU's set. A software developer usually does not explicitly use the instructions since the compiler or interpreter automatically does that.

In this work we are focusing on the first class of co-processors, also called external co-processors. In Chapter 6 we will address the second class, which is at the moment often referred to as heterogeneous processor architecture.

## 3.3. Software Frameworks and Libraries

Co-processors for general-purpose calculations require special software to use them. Developers have to decide whether they want to use a framework to write low-level-code or libraries that provide functions for a certain domain. When programming for NVidia GPUs, which are used in all experiments in this thesis, one has the choice between using OpenCL and CUDA. The advantage of OpenCL over CUDA is that not only other co-processors but also the CPU is able to execute it. We conducted some experiments that show that we can run OpenCL code developed for the GPU on the CPU without changes (see Section 5.5.3). Although this might be slower than a native parallel CPU implementation, it is still faster than sequential execution. Next to CPUs, other GPU vendors also support OpenCL. AMD even stopped developing *Stream*, which was their specific framework for General Purpose Graphics Processing Unit (GPGPU) computing. Intel also provides OpenCL for their GPUs and for the Xeon Phi. We expect future devices from different vendors to use OpenCL as well, because it is an open specification and many applications already support it.

Both frameworks focus on parallel architectures and are very similar in principle. The code running on the GPU—more general the device—is called kernel and uses the programming model of workgroups and threads as explained above. OpenCL and CUDA are both dialects of the C programming language. In many cases it is easy to transform code between CUDA-C and OpenCL-C, because it is mostly a matter of translating keywords. For example, synchronizing threads is done by `__syncthreads()` in CUDA and `barrier(CLK_LOCAL_MEM_FENCE)` in OpenCL. In general CUDA provides more language features, such as object orientation and templates, as well as more functional features. As long as the missing functional features in OpenCL do not limit the algorithm the runtime of OpenCL and CUDA kernels for the same purpose is comparable in most cases. According to the authors of [53] OpenCL is mostly slower (13% to 63%) on NVidia GPUs, but they found that often there is no difference in performance at all. However, sometimes kernels can be designed in another way with features only available in CUDA, such as Dynamic Parallelism and dynamic memory allocation. The first one allows to spawn kernels from inside other kernels and the synchronization of all threads on the device and the second feature allows developers to allocate memory inside kernels. This opens up new possibilities, which might affect performance as well (see Section 3.4.1).

The conceptional difference between OpenCL and CUDA is the way, kernels are compiled. CUDA has to be compiled by NVidia's *nvcc* and the object files are then linked to the CPU object code. To support future devices the compiler allows to include an intermediate code, called PTX, which is generated by nvcc at compile time and then used to produce the binary code for the specific device by the CUDA run-time. Because OpenCL targets a wider range of devices, it makes no sense to deliver device-specific code and ship it with the binaries in advance. Instead the OpenCL code is compiled at run-time by the driver of the device. In Chapter 5 we use this feature for our query execution framework.

In Section 3.1.2 we already mentioned that there are details about the architecture, that are performance critical when unknown, e.g., coalesced memory access or under-

utilization. There are many other pitfalls in GPU-programming, but similar to CPU software there are already libraries that provide optimized algorithms for certain domains.

**Thrust** Similar to the Standard Template Library (STL) in C++ the Thrust library provides basic algorithms to work with vectors of elements. It provides transformations, reductions, and sorting algorithms for all data types available on the GPU.

**CUBLAS** Algorithms for the domain of linear algebra is available for NVIDIA devices in form of the CUBLAS. As the name suggests the library provides the common Basic Linear Algebra Subprograms (BLAS) interface to handle vectors and matrices. The CUSPARSE library provides similar algorithms for sparse matrices and vectors.

**CUFFT** The CUFFT library provides the CUDA version of Fourier transforms, which are often needed in signal processing

**NPP** The NVIDIA Performance Primitives are a library for image and video processing. In contrast to the other libraries, where every function spawns at least one kernel, the functions of the NPP can also be called from within kernels.

**unified SDK** The BLAS and FFT primitves are also available in OpenCL as part of the unified SDK. It also includes the former Media SDK with video pre- and post-processing routines.

Another way to use parallel co-processors without handling the details of the underlying hardware is OpenACC [79]. With OpenACC developers can use patterns to give the compiler hints on what can be parallelized and executed on the GPU. Listing 3.1 shows code for the matrix multiplication in C with OpenACC. The `pragma` in line 1 tells the compiler that the following code block should be executed on the co-processor, a and b are input variables, and c has to be copied back to the host after kernel execution. By describing the dependencies between the iterations of a loop, the compiler can decide how to parallelize execution. `Loop independent` in lines 3 and 5 encourages to distribute the execution over multiple cores while `loop seq` forces the execution of one loop to one core, because every iteration depends on the previous one.

Listing 3.1: Matrix multiplication in OpenACC

```
1 #pragma acc kernels copyin(a,b) copy(c)
2 {
3 #pragma acc loop independent
4  for (i = 0; i < size; ++i) {
5 #pragma acc loop independent
6   for (j = 0; j < size; ++j) {
7 #pragma acc loop seq
8    for (k = 0; k < size; ++k) {
```

```
9       c[i][j] += a[i][k] * b[k][j];
10     }
11    }
12   }
13  }
```

## 3.4. Micro-Benchmarks

In research, magazine articles, blogs and tutorials [82, 8, 23] are a lot of "do's and don'ts"can be found when it comes to GPU processing. One of the ubiquitous claims is that a problem must be "large enough" to justify offloading to the GPU. Hence, small problem sizes should not be calculated on the GPU. Another "don't", which is especially important for problems in the DBMS domain, is that you should not process strings on the GPU. In this section we present some simple micro-benchmarks we designed to evaluate these claims.

There are three reasons why "small" problems are always calculated faster on the CPU:

1. Starting a kernel requires the CPU to push the code to the GPU and transfer the kernels parameters before the calculation can begin.
2. Since we use the external GPU, we have to transfer input data to the GPU and the results back.
3. Because of the parallel architecture, the workload has to be large enough to be distributed over all cores.

In Section 3.4.1 we conduct some experiments to measure the overhead for the first two points. To overcome the transfer problem (point 2), it is often suggested to overlap transfer and computations. We show how this method can be used and what its implications are in Section 3.4.3. Although many problems can benefit from the GPU's parallel architecture, there is a break-even point in workload size, at which the GPU can be fully utilized utilized (point 3). We try to get a feeling for this break even point in Section 3.4.4 on the example of matrix multiplication. Finally, in Section 3.4.5 we take a look at the claim that the GPU is not good at string processing.

### 3.4.1. Memory Bandwidth and Kernel Overhead

When involving external co-processors for calculations there is always some overhead for transferring the data needed and calling the kernel.

**Transfer** In the first experiment we copy data from the main memory to the device and compare the two different modes that are available in CUDA (and OpenCL as well). The default transfer is done on page-able data in main memory, i.e., we can transfer data from any memory segment accessible by the process. In this mode the CPU pushes data

to the GPU after checking if the page is in main memory at the moment or swapped to disk. The second mode transfers data from pinned memory to the GPU. The operating system guarantees that a pinned page is always in main memory and will not be swapped at any time. Therefore the GPU can transfer the data without interaction of the CPU with Remote Direct Memory Access (RDMA). To use pinned transfer, the process has to allocate pinned memory or pin already allocated memory. Then the process tells the GPU's copy engine to transfer data from a given memory segment and continues with other operations. We can see in Figure 3.3 on the left that copying from pinned memory is always faster than the default operation. What is often concealed is that pinning memory involves an expensive system call. As shown in the Figure the Pin call takes at least 300 µs, which is approximately the time needed to transfer one megabyte of data. For larger memory segments it takes almost the same time like copying the data right away. Therefore, it makes no sense to naively pin memory to transfer data only once. In Section 3.4.3 we discuss how pinned memory can be used instead for streaming to avoid the costs for pinning large amounts of memory. In Section 5.1 we take another look at different solutions for the transfer problem.

**Kernel Invocation** Starting a kernel requires the CPU to initialize a connection with the GPU via PCIe bus. The GPU then has to reserve resources and prepare the environment for the kernel execution. Compared to a function call on the CPU this is very expensive. NVidia's new GPU architecture "Kepler" introduced a new feature called "Dynamic Parallelism" that allows to start kernels from inside other kernels and synchronize all threads on the device without the CPU's interaction [77].[3] We conducted a small experiment to check if we can gain performance from this new feature. In the experiment we pair-wise multiply the elements of two arrays containing 100 mio double-precision numbers. Every thread reads two numbers from the arrays, multiplies them, and writes the result back to a result array. Usually we would start one kernel and choose the number of workgroups and threads so that one thread had to multiply only a few elements (in Section 5.5.4 we evaluate a good number of threads to start). For the sake of our experiment we split the workload and start multiple kernels. Every kernel now starts one thread per pair, i.e., every thread executes exactly one multiplication. We wait for each kernel to finish and do a global synchronization before we start the next one. The more kernels we use the more overhead for invocation and synchronization will be necessary. For the traditional approach we start the kernels directly from the CPU. With Dynamic Parallelism we start one initial kernel with one workgroup and one thread that starts the multiply-kernels. Naturally, we would assume that Dynamic Parallelism is faster because no communication between CPU and GPU seems to be involved when new kernels are started.

Figure 3.4 shows that we need around 16 ms to execute the task with 10 kernels, each having 39 062 workgroups. Note that we start a thread for every single multiplication. Independent of the approach, using a high number of kernels slows the process down. The high execution time for more than 10,000 partitions/kernels is mostly due to the

[3] At least there is no visible interaction from a developer's view.

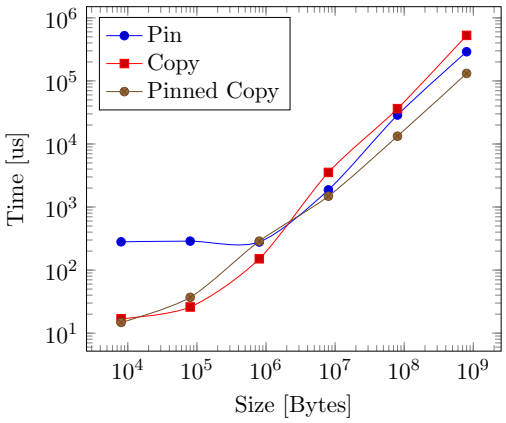



Figure 3.3.: Comparison transfer pinnend and pageable memory

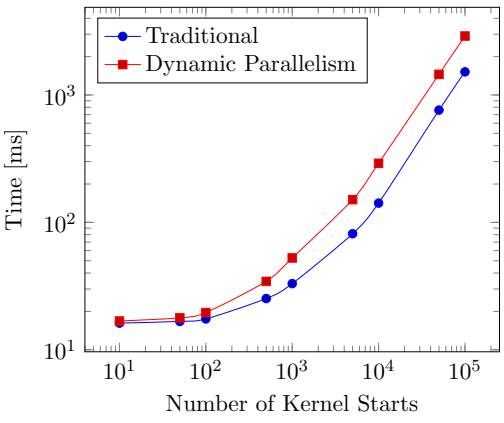



Figure 3.4.: Traditional kernel starts vs Dynamic Parallelsim

underutilized GPU. In this case every kernel starts 40 workgroups with 256 threads each to do the work. Although 10 000 threads should be enough to utilize all cores, most of them have to wait for I/O because of the high latency of the GPU's RAM. More threads are better to hide this latency. If this was the only reason, there should not be a large overhead for 500 partitions, where every kernel starts around 800 workgroups. Still the calculations take more than 10/18 ms (traditional/Dynamic Parallelism) longer than just using one kernel, which can only be caused by synchronization and overhead for starting new kernels. We can also see that it actually takes longer to start kernels from the GPU with Dynamic Parallelism than on the CPU with the traditional approach. For 10 kernel starts the whole process takes around 600 µs longer with Dynamic Parallelism and 10 kernels, the absolute difference for 100,000 kernels is around 1.4 s. So we can draw two conclusions from this experiment: first, global synchronization should be avoided and the less kernel calls are needed in the algorithm the better. Second, in its currenct state Dynamic Parallelism does not change performance for the better. At least not, if we use one thread on the GPU to fulfill the role of the GPU as we do here.

### 3.4.2. Single Core Performance

GPUs may have many cores, but these cores are very weak compared to a CPU core. For the following experiment we executed the naive matrix multiplication algorithm shown in Listing 3.2 on the CPU (Intel Xeon E5-2665 at 2.40 GHz) and on the GPU (K20 at 732 MHz). [4]

We do not measure transfer but pure execution time, Figure 3.5 shows the execution time for multiplying two square matrices. The CPU is approximately 250 times faster, independent of the size. Although we cannot just conclude that a CPU core is 250 times faster, as there are other parameters such as cache and memory bandwidth that play a

[4] See Appendix A.1 for Details on Hardware.

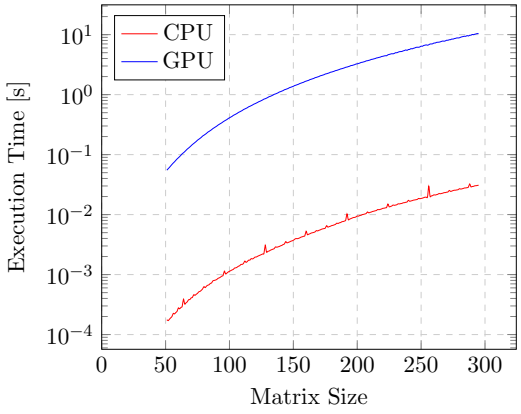



Figure 3.5.: Execution time for matrix multiplication on a single core

role, but the experiment shows the importance of using parallel algorithms and as many cores as possible to get a good performance on the GPU.

Listing 3.2: Naive algorithm for matrix multiplication

```
1 for(uint i=0;i<size;++i) {
2   for(uint j=0;j<size;++j) {
3     for(uint k=0;k<size;++k) {
4       C[i+j*size] = C[i+j*size] + A[i+k*size] * B[k+j*size];
5     }
6   }
7 }
```

Of course this experiment says nothing about the ability of GPUs to multiply matrices, any other sequential algorithm would have shown a similar behavior. In Section 3.4.4 we show the performance for GPU and CPU when optimized algorithms are used on multiple cores.

### 3.4.3. Streaming

In Section 3.4.1 we discussed that pinned memory transfers have a higher bandwidth but require the time-consuming allocation of pinned memory on the CPU. So by just pinning memory segments before transfer, there is no benefit. The real advantage of pinned transfers is that they can be done asynchronously, i.e., the CPU can continue working while the GPU does the transfer. Additionally, the PCIe bus supports full-duplex communication and modern GPUs have two copy engines. Hence, we can copy data from and to the GPU simultaneously at full bandwidth. To gain advantage from this we have to change the way we do uploads to and downloads from the GPU's memory. Instead of allocating pinned memory for all of the data to be transferred at once, the 4-way-concurrency pattern can be used to achieve the same high bandwidth with a small pinned memory buffer [93].

We explain and evaluate this pattern by taking a look at a very simple algorithm that is part of the Delta Merge process explained in Section 2.2.4. In SAP HANA every distinct value of one column is stored in a sorted dictionary and the values in the column are references to the entries in the dictionary. The Delta Merge rebuilds the dictionary in a first step and updates the references in a second one. We concentrate on the second step which uses a mapping from the old reference to the new reference to update the column. Since references are just integers, the mapping is an array, where each new reference is stored at the position of the old one. Therefore, the algorithm just iterates over the vector of references, and replaces each element with the mapped value as shown in Listing 3.3.

Listing 3.3: Second part of the delta merge algorithm

```
1 //src and dest can point to the same address
2 void recode_cpu(int *dest, const int* src,
3                          const int *map, size_t size) {
4     for(uint idx=0;idx<size;++idx) {
5         dest[idx] = map[src[idx]];
6     }
7 }
```

For a parallel implementation on the CPU we can simply use OpenMP's `parallel_for`. There are no dependencies between the loop iterations, so OpenMP starts enough threads to saturate all CPU cores and distributes the work evenly. The parallel GPU implementation does the same, but because we want to use streaming, a bit more effort is necessary: first, we copy the map to the device memory. Afterwards, processing and streaming of the reference vector overlaps with the help of the 4-way-concurrency pattern. We allocate three pinned buffers in the CPU's RAM and three corresponding buffers in the device memory. Now we do four things at once as depicted in Figure 3.6:

- the CPU copies references to the first buffer
- the GPU's copy engine copies references from a second buffer to the GPU's memory (first buffer there)
- the GPU processes the references in the second buffer
- the GPU's second copy engine copies the already processed elements from the third buffer on the GPU to the third buffer in main memory

Additionally, we copy the elements from the third CPU buffer to the target memory as part of the first step.

In Figure 3.7 we can see the throughput (on the Z600[5]) for a vector with 4 GB of integers depending on the size of the mapping array. An 1 kB array holds 256 different integers, i.e., there are 256 different entries in the column's dictionary. The references are

[5] See Appendix A.1 for Details on Hardware.

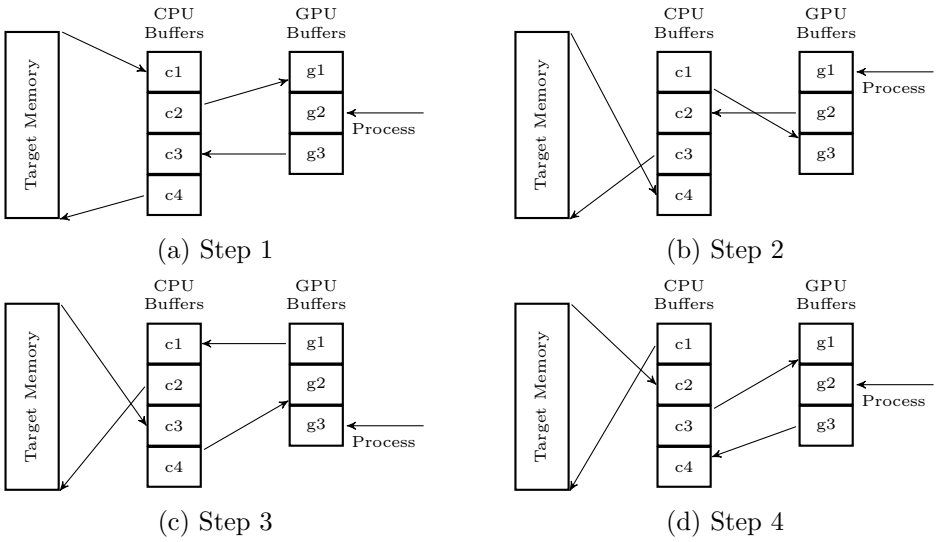



Figure 3.6.: Streaming with the 4-way concurrency pattern

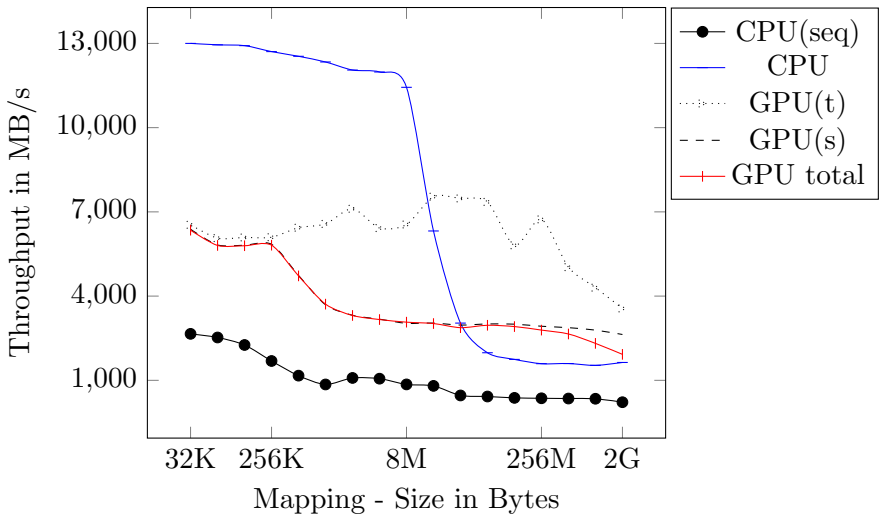



Figure 3.7.: Streaming on the example of the recode algorithm

uniformly distributed over the column vector. The blue and the red curve are the ones to compare: blue is the CPU using all its cores, red is the GPU. We also added the sequential CPU implementation *CPU (Seq)* as a baseline. *GPU (t)* shows the throughput for only transferring without executing the kernel and *GPU (s)* is the GPU process without the initial transfer of the mapping array.

The throughput for small mapping arrays is higher, because the array itself fits into the CPU's caches. Therefore, only the reading and writing of the column vector requires access to slower main memory. Since every value in the column vector is touched only once, reading and writing cannot benefit from the cache.[6] The larger the mapping array gets, the more access to slower caches (L2 and L3) is needed. As we can see 13 GB/s is the upper limit for reading the column vector from memory and writing it back. At 8 MB the array does not fit into the L3 cache anymore, and a third random memory access is needed every time a part of the mapping array is not in cache. At more than 128 MB the cache is practically useless; because of the uniform distribution every access to the mapping array is a cache miss. In contrast to the sequential reading and writing of the column vector the random access is much slower. The throughput is constant at 1 GB/s.

The GPU's streaming throughput is around 7 GB/s. Until 256 kB there is no difference between *GPU (s)* and *GPU total*, i.e., the GPU processes a segment faster than it transfers the next one. In these cases the mapping array fits into the GPU's Cache. With larger arrays we have more cache misses and execution takes longer than transfer (just streaming the data—*GPU (s)*—is independent of the array size). For array sizes larger than 128 MB the GPU is faster than the CPU, because both devices cannot use their caches and random access to the GPU's memory is faster. If the array becomes larger than 256 MB the throughput on the GPU decreases and the array's initial transfer becomes the bottleneck.

We can show here that data intensive algorithms can benefit from the high bandwidth on the GPU if they are stream-able and the caches on the CPU get to small to hold data that is periodically needed. However, this example is certainly artificial. Scenarios, where dictionaries contain more than 20,000,000 entries and these are uniformly distributed are not found often. Also, references are usually bit-packed/compressed. Therefore, the re-encoding requires more computation and less transfer [104]. It is hard to predict the effect on our comparison, because the CPU algorithm is also memory bound and will therefore benefit from the compression as well.

### 3.4.4. Matrix Multiplication

Operations on large matrices are often said to benefit from execution on the GPU. However, for every operation there is a break-even point at which it starts to make sense to use an external co-processor. For small matrices the overhead to start a kernel on the GPU is too high and because of the parallel architecture the calculations cannot utilize all cores on the GPU and therefore use only a fraction of the available power.

[6] In fact, a good implementation should use an instruction to bypass the caches for reading and writing the vector.

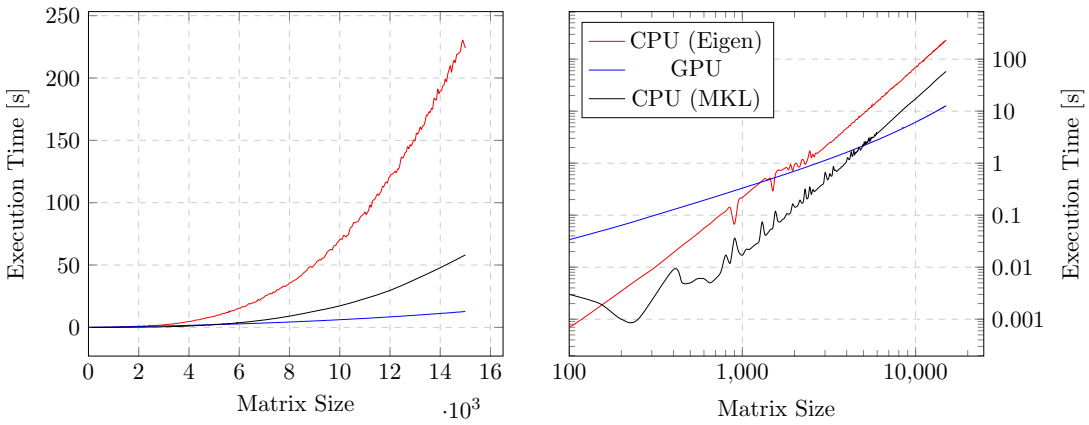



Figure 3.8.: Matrix multiplication on CPU and GPU

On the example of square matrix multiplication we compare three libraries for linear algebra to find the break-even point. For the experiment we use the K20 GPU and measure transfer of input and result as well as the execution time. [7] On the CPU we use the single-threaded Eigen library[8], the multi-threaded Intel Math Kernel Library (MKL), and on the GPU the CUBLAS library, which is part of the CUDA toolkit. The results are shown in Figure 3.8 in linear scale on the left and logarithmic scale on the right. Although the Eigen library uses only one core for calculations on the CPU, it is faster than the GPU for the multiplication of matrices smaller than approximately $2500 \times 2500$ double precision elements. The MKL, which saturates the 12 CPU-cores during calculation, is up to 10 times faster than Eigen for multiplying matrices smaller than $2000 \times 2000$. For larger matrices the speedup is around 3 to 4. At a size of $5000 \times 5000$ CUBLAS is faster than both CPU implementations. If the matrix is even larger the speedup of the GPU compared to the CPU is dramatic. The computation of $15000 \times 15000$ elements on the GPU is 20 times faster than on the CPU with Eigen and 4 times faster than the MKL.

### 3.4.5. String Processing

In the last experiments we have seen that the GPU is well suited to handle data types such as integers and floating point numbers. In this section we want to evaluate how the GPU handles strings. An indicator that GPUs may not be good at this is that no official library is able to work with strings. Even very simple string-operations, such as comparisons, are not available and have to be implemented by the application developer.

There are two main problems when dealing with strings: first, variable length values do not fit well to the SIMD processing architecture of GPUs. There is always a fixed number of processing cores doing the same instruction. We cannot dynamically assign

[7] See Appendix A.1 for Details on Hardware.
[8] http://eigen.tuxfamily.org/

work to each of them, but variable length values require exactly this. Second, Strings—even with constant length—are data intensive and require only a very small amount of processing in general, e.g., a string comparison often requires just the comparison of the first character. The transfer to the GPU therefore often dominates the processing time.

A popular suggestion to cope with the problem is dictionary encoding. If there is only a small number of distinct values, this limits the amount of actual string operations to a minimum. In case of a lot of unique values, we can process the dictionary itself on the CPU and the references on the GPU. This leads to the situation that either GPU and CPU have to communicate every time a string operation is needed or that the CPU has to pre-process everything, which can easily lead to unnecessary overhead. An equi-join on a string column for instance would mean that the CPU has to prepare a mapping for the string references in the joined columns, but in most cases the GPU may only need parts of these mappings. Hence, dictionary encoding with large dictionaries only makes sense if it is the minor part of the workload, otherwise the GPU will barely be used.

To evaluate these thoughts we compared the CPU/GPU performance of two operations which are often used on databases with columns containing strings. The first operation checks how many strings in one column of a column-wise stored table start with a given string (*startswith*-operation). The second operation searches all values for a sub-string (*substring*-operation) and returns the number of values that match.

For both operations we implemented three different versions:

- **std:** In the first approach we used the containers of the C++ standard library. The column is a simple vector of strings. Since there is no library available which handles strings on the GPU, this was only tested on the CPU. It is our baseline.
- **Custom:** Our hand-written implementation uses one block of memory to store all strings contiguously. An array holds the information on where one value starts and how long it is (of course one of these variables would be enough, but for the GPU this gives a few advantages). The core of the substring operation is the Boyer-Moore-Horspool algorithm. We tried different implementations and decided to use the OpenSource implementation from `https://github.com/FooBarWidget/boyer-moore-horspool`, because it shows the best performance. As a side note: we had to implement basic functions such as `strstr()` for the GPU on our own and used them for both implementations. Interestingly our `strstr()` is around 20% faster than the C library's, because—like `memcmp()`—it does only check for equality not for greater/lesser.
- **Dictionary:** We used our Custom structure as dictionary and a simple array holds the references to the dictionary. We implemented it as read-only structure, i.e., the strings in the dictionary are sorted like in SAP HANA's main storage.

These three implementations of the two operations are now executed with three different settings:

- Single: The sequential CPU implementation is straight-forward. With one thread we just compare every value with the given string in a loop.

- Parallel: The parallel implementation uses OpenMP's `parallel for` to execute the loop on all available cores, which requires one additional line of code.
- GPU: Although the basic execution logic on the GPU is the same—a high number of values is compared to the search string in parallel—the implementation requires more effort.

The process can be split up into 6 different steps:

1. allocate memory on the GPU
2. transfer the data to the GPU
3. execute the kernel
4. transfer the result back to the CPU
5. process the result on the CPU
6. free the memory on the GPU

Step 5 is necessary, since the kernel does not give the final result—the number of matching values—but a bitvector with a bit set on every matching position. With a second kernel call we could add these numbers up to the final result, but this would lead to much overhead on little work. Hence, we process this on the CPU.

As data set we used the L_COMMENT column of the TPC-H benchmark (SF 10), which consists of 60 mio strings with 10 to 43 characters. The column consists of approx. 70% distinct values. For our experiment we searched for the term "the". 935 863 entries start with this term and it is a substring of 2 068 744 values. Since transfer plays an important role for this experiment, we used the test machine with the K10 GPU and PCIe Generation 3. [9]

**Results Startswith** In Figure 3.9 on the left-hand side we can see the results for the Startswith operation. The single core *std*-implementation of the substring implementation takes around 600 ms to execute (STD SC). With the help of OpenMP we could achieve a speedup of about 4 by executing the loop in parallel (multi-core—STD MC). The *Custom*-implementation runs approximately in half the time for the sequential as well as the parallel implementation. On the GPU bar we marked the different phases. It shows that 90% of the time is needed to allocate/free memory (named GPU overhead) and transfer the data to the GPU, less than a milli-second is spent on the CPU to process the interim result. The overhead is mostly necessary for allocation and transfer of the input data. The allocation of memory for the interim results and its transfer take approx. one ms and are therefore negligible. Because of the high transfer costs the GPU implementation needs three times as long as the parallel CPU implementation, although the kernel is four times faster. The operation on the dictionary encoded column are two simple binary searches on the dictionary to find the range of values that match the term we search for. 99% of the time is spent to find the matching values in the array.

[9] See Appendix A.1 for Details on Hardware.

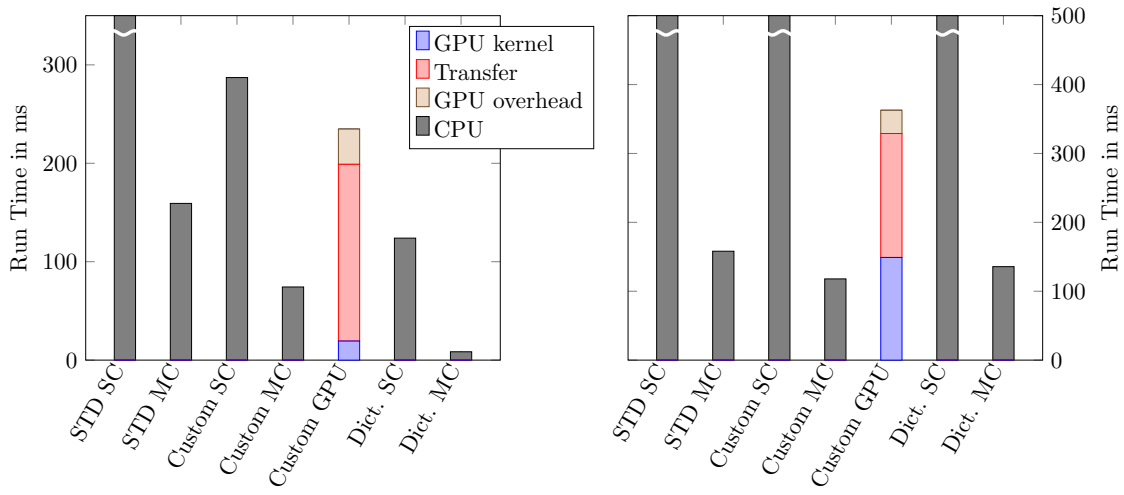



Figure 3.9.: The startswith and the substring operation

**Results substring** The single-threaded execution of the second operation takes between 2 and 3 seconds for the three different implementation. The focus of this experiment is the comparison of multi-core CPU and GPU, therefore the plot is zoomed to the interesting results. The plain execution time for all variants with CPU and GPU lie between 110 and 150 ms. However, the necessary transfer to the GPU adds 200 ms again.

The *startwith* operation is much faster on the dictionary encoded column because it benefits from the sorted dictionary. In contrast, dictionary encoding is disadvantageous for the *substring* operation, because a full scan is necessary in the dictionary first to get the matching references and then another lookup is necessary to find the references in the array. The more matching references are found in the dictionary, the more time is necessary for the lookup operation. If there were no matching values in the column the operation would most likely be a bit faster on the dictionary encoded column. The same counts for small dictionaries. The startswith-operation is clearly memory-bound. We can only achieve a speedup of around 4 with 16 cores. There is a significant difference between the std and the custom implementation, because the contiguous memory layout allows much faster access while the usage of std::string forces the algorithm to dereference another pointer. The substring-operation is obviously CPU bound. We achieve a speedup of 20 with 16 cores resp. 32 threads. The Horspool algorithms reads a string more than once. Therefore the cores operate on values in the cache and are not limited by the memory bus. When executing the operations on the GPU the transfer dominates the execution time, even if we transfer only the dictionary-encoded values. However, it is a surprise that the plain execution of the startswith-operation is significantly faster on the GPU than on the multi-core CPU. We think this is because—although the code does not show it—the actual execution fits well to the SIMD processing model. All threads compare the first character with the constant value at the same time and then in most

cases the loop continues to the next iteration because there was no hit. This is not the case for the substring operation, which is indeed slower on the GPU than on the CPU. The reason for this might be the high branch divergence when executing the Horspool algorithm. Another problem is that we cannot use the faster Shared Memory for our implementation because it is shared between the threads of one workgroup and every thread needs a different amount because of the variable length values. Dealing with this problem would induce more overhead for the memory management and more branch divergence because every thread would execute different code depending on where the string is stored. Maybe just comparing all values in parallel is not the right approach, but a massively parallel SIMD-like algorithm for searches in short strings may not be possible.

The claim, that GPUs are bad at string operations is only half the truth. Compared to a sequential CPU implementation the GPU is factors faster. In case of the startswith operation the GPU kernel alone is even faster than 16 cores on the CPU, which is surprising. However, there are two problems. First, we cannot ignore the transfer, which is the major bottleneck for string operations. Second, we used read-only data structures for that experiment. The implementation of a std-like vector of strings requires massive pointer-handling (one for every string). These pointers can only be created on the CPU, i.e., every string must be allocated and copied by the CPU. De-referencing these pointers on the GPU is also slow, because of the high latency of the main memory. In both operations the dictionary approach is not beneficial. Transferring a vector of integers to the GPU just for a simple comparison makes even less sense than processing the strings there. In most scenarios string processing should indeed be done one the CPU.

## 3.5. DBMS Functionality on GPUs

Because of the raw calculation power of modern GPUs, they provide new opportunities for CPU bound tasks in MMDBMS. However, many operations in a MMDBMS are still memory-bound, and cannot benefit from a more powerful co-processor, especially with the PCIe bottleneck. Also, as we explained in this chapter that not every task is suitable for GPU-execution. String processing and data-intensive tasks usually fail because of the transfer bottleneck. In other cases the workload has to be large enough to fully utilize the parallel architecture and to justify the overhead for starting a kernel.

Based on what we discussed in the last two chapters we draw a conclusion, how we can speed up a DBMS with the help of a GPU. In a general DBMS we find four different categories of tasks—depicted in Figure 3.10—where a gain in performance would have a significant impact on the whole system, i.e., reduce the time users have to wait for answers to their queries.

**Application Logic (1)** Most DBMS provide the possibility to integrate application logic in form of *Stored Procedures*. This way the DBMS can execute logic that is written in another language than SQL. C++ or Java might be simpler, provide more possibilities,

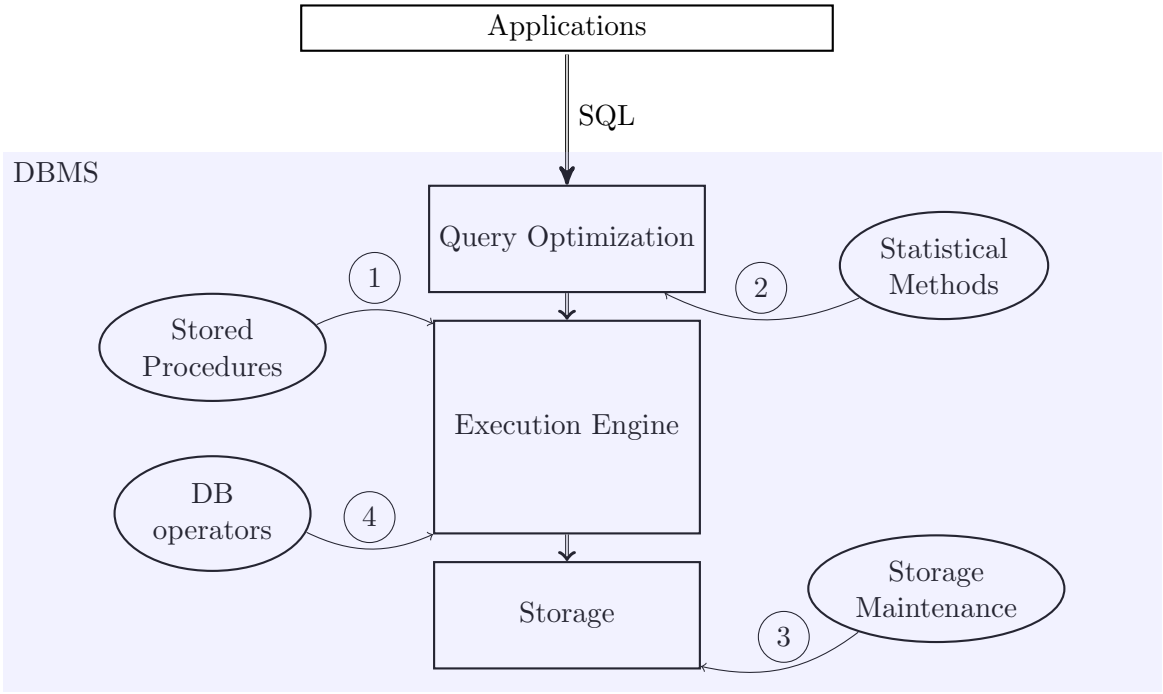



Figure 3.10.: Possible co-processing tasks in a DBMS

or more performance in some cases. The question is if and how we can use co-processors to benefit from tasks of this first category.

**Query Optimization (2)** Users and applications usually access or modify the database through SQL, which is a declarative language, i.e., it does not specify how a query has to be executed. Hence, a DBMS first has to build a Query Execution Plan (QEP) that tells the execution engine which operators to execute in which order. The performance of the execution strongly depends on this plan. So the second category consists of all tasks necessary for building and optimizing the QEP.

**Storage Maintenance (3)** Another important factor for the performance of the query execution is the accessibility of the stored data. In general, storage aims for three targets: a reduced memory footprint, fast reading, and fast modifications of the data. It is not possible to optimize for all three. Indexes, for example, speed up reading/searching the data, but need memory to be stored and have to be adjusted, when the data is modified. Compression reduces the memory footprint, but modifying compressed data requires additional effort. Usually a DBMS partitions the data in a way, so that frequently modified data is stored in a different form than data that is only read. Data that is accessed only seldom is archived, i.e., it is heavily compressed. The system moves data automatically to the right partition based on usage, load and requirements of the system.

The faster this maintenance can be done, the more often we can do it. Additionally, if parts of this maintenance can be offloaded to a co-processor, resources on the CPU are free for other operations.

**Query Execution (4)** Our last and maybe most important category is the execution of the QEP itself. Typical database operators such as joins, selections, and aggregations have different characteristics and not every one is suitable for the GPU. Also, there are different ways on how data flows from one operator to the next.

### 3.5.1. Integrating the GPU for Static Tasks into the DBMS

The CPU is a programmable universal processing unit in our computers. It can process images, filter big amounts of data by certain criteria, calculate scientific formulas on different representations of numbers (fixed point and floating point), analyze texts, and everything else that is done with computers today. Since the CPU's purpose is so generic, it cannot be efficiently optimized for one task. A co-processor in contrast is optimized for a subset of tasks, e.g., graphic processing in case of the GPU. It is not well suited for operations that do not match the SIMD pattern, such as string processing. Hence, we do not aim to build a DBMS that runs completely on the GPU, but try to identify tasks in a DBMS suitable for GPU processing.

There are a number of tasks in a DBMS that can be considered independent from the rest of the system. Therefore, it is possible to offload these tasks for computation to the GPU without conceptual changes to the DBMS. The general pattern to distribute a task in a parallel co-processor architecture is depicted in Figure 3.11. Phases one and five are necessary because the GPU cannot access the CPU's memory—they represent the transfer bottleneck. Phases two and four are part of the necessary overhead for parallelisation, resp. the distribution of work over different cores.[10] Phases of this pattern can overlap with other phases, partitioning can for instance be part of the transfer. In some cases it might also be necessary to repeat phases, e.g., for iterative algorithms such as the tasks presented in Sections 4.2.2 and 4.1.2.

[10] Partitioning and merging are also necessary for parallel implementations on a multi core CPU.

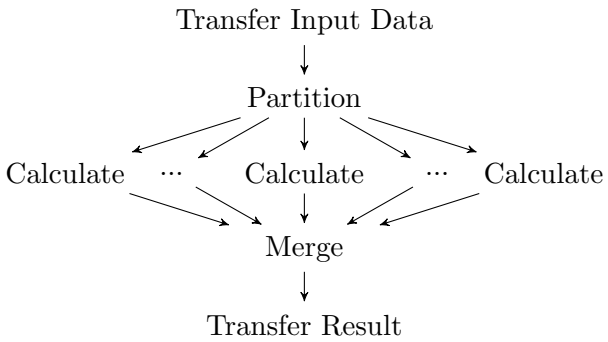



Figure 3.11.: A general pattern for running tasks on the GPU

There is no guarantee that a task runs faster on the GPU. A general rule is that computational intensive operations may benefit from offloading, i.e., the transfer bottleneck is not a problem, because a lot of processing is done on a small amount of data. However, there are number-crunching algorithms that do not fit to the GPU's programming model [62]. Although the data transfer is not a problem, the CPU can solve the problem faster and more efficiently. On the other hand, there are data-intensive problems that can benefit from the GPU's characteristics (mainly the high bandwidth on the device). The small experiment of Section 3.4.3 shows that it is not easy to predict the performance of an algorithm. But if certain criteria are given, there is a chance that we benefit from offloading the task to GPU.

Criteria that indicate a performance gain when using the GPU:

1. There is an algorithm for the task that works in parallel.
2. There is an algorithm for the task that works massively parallel in a SIMD fashion.
3. Processing the data on the CPU takes longer than transferring it to the GPU.
4. There is no need for additional transfers (swapping to main memory) in phase three.
5. Transfer to the GPU and actual computation can overlap.

These five points can be categorized: the first two points deal with the suitability of a task to the GPU's architecture while points three to five consider the transfer bottleneck.

As we already explained in Section 3.1.2 we need to adjust processing to an architecture with two levels of parallelism. Point one means that we first have to find a way to execute the algorithm in parallel, whether this is on the CPU or on the GPU plays no role at this point. In some cases this parallel execution already works in a SIMD fashion. Again, Section 3.4.3 is an example for this. However, often it is necessary to introduce a second level of SIMD parallelism (point two). In Section 4.3 we explain the difference in detail by porting a simple sequential algorithm to the GPU. For this point the used data types are also crucial: mostly, short strings or other variable-length types cannot be processed efficiently in a SIMD fashion.

Point three is the most basic check for external co-processors and can be easily calculated. We know the theoretical bandwidth of the PCIe bus and can calculate how long it takes to transfer the input data for the task. No matter how good the algorithm executed on this data is and even if we can perfectly interleave calculations and transfer: if the transfer is slower than the CPU there is no chance that there can be any benefit. Tasks with that characteristic are simply data-intensive. We take a look at one example in Section 4.3. GPUs are sometimes proposed as a (part of the) solution for *Big Data* problems, but most of these problems are actually memory bound and cannot benefit

from the GPU as long as there is the transfer bottleneck.

Point four does not mean that data larger than the GPU's memory cannot be processed efficiently (although it may be a first sign...), but the algorithm should be able to transfer a partition of the data, process it, and proceed to the next one. If you repeatedly need to transfer the same data back and forth in phase 3 because the free memory is needed in between (known as swapping), there is a good chance that there will be no benefit in using the GPU; or any other external co-processor.

If, however, the data can be processed while it is transferred (point five), and processing is faster than transferring, the bandwidth is the only limit. So even if the CPU's throughput is just slightly lower than the PCIe bus', there is a chance that the whole process is faster on the GPU. Our micro benchmark in Section 3.4.3 is such an example. In Chapter 4 we identify DBMS tasks that can be ported to the GPU based on these rules. We focus on how they can be integrated into the system in a modular way.

### 3.5.2. Re-Designing Dynamic Tasks for Co-Processors

The tasks of the categories (1), (2), and (3) have a very important characteristic in common. The workload to be processed can be called static: their input changes only in the size of the data and parameters that influence the number of calculations per byte needed. The tasks themselves are static as well: they are built by combining primitives, e.g., by sequentially executing mathematical operations on matrices. These primitives are usually executed in a known order and independent of the whole workload size, the execution time for each primitive in comparison to the others can be predicted. In case of matrix operations, there are for instance additions, which are uncritical compared to a following matrix inversion. When trying to optimize such static tasks, the developer looks for the *hot-spot operations*, where most time is spent, and optimizes these. When using a co-processor, in many cases it is beneficial to just execute these operations on the co-processor and leave the minor operations to the CPU.

Another technique to optimize a task is to merge the primitives in a way that there are no borders resp. synchronization points between them any more. In Section 3.4.1 we already discussed the negative impact of kernel starts on performance. Therefore, it is crucial to put as much work as possible into one kernel call. It may even be possible to merge small operations into a larger operation, which benefits from offloading. This of course destroys maintainability and modularity of the program itself and is only possible, because we know the order or primitives and with this certain characteristics of there intermediate results.

Dynamic tasks consist of modular primitives that are executed following a plan given at run-time. Without additional analysis we cannot predict which of these primitives is the hot-spot operation. In Section 3.4.4 we have seen that the GPU is factors slower in executing multiplications on small matrices. If we do not know the input size for operations like this in advance and they are executed repeatedly, the whole task might also be factors slower. Since a primitive can only be a kernel at first glance, there is no possibility to merge them. Therefore, the plan-optimizer must be able to predict intermediate result sizes and schedule operations accordingly. And, it has to do the

low-level-optimization as well. While the CPU itself optimizes the execution of instructions, the processing units on the GPU do almost no optimization. This is typical for co-processors: the architecture is exactly designed to solve problems of a certain domain and every transistor is placed to provide as much calculation power as possible. Therefore, operations have to be developed and optimized to the instruction-level. In case of GPUs there is the additional challenge of considering the memory hierarchy and schedule memory access and calculations correctly. Again, with kernels as primitives we can use this memory hierarchy only inside. Even if results were small enough to keep them in shared memory, they cannot be exchanged without transfer to the devices main memory.

All these reasons show that the approach of having modular kernels as building blocks is not going to be beneficial for dynamic tasks. So instead we propose to generate and compile the kernels at run-time. This way, we can make use of the memory hierarchy across primitives and merge them to achieve a small number of synchronization points during execution. The compiler itself can do the low-level optimization. For the dynamic task of query execution we explain this in Chapter 5.

### 3.5.3. Scheduling

While there are tasks that can benefit from the GPU as processing unit, most of them only work well for certain inputs or certain parameters. If on the one hand the data to be processed is too small, only some of the GPU's cores can be used and the overhead for starting a kernel and parallelizing the execution dominates the run-time. If on the other hand the input is too large to fit into the GPU's memory, it may be necessary to transfer input data and intermediate results multiple times from main memory to device memory and back. In these cases, the CPU should be used for calculations instead. The break-even points change for every task and every hardware-combination. Especially, when the execution time depends on multiple parameters such as the data type of the input or the number of iterations, determining this point manually is not feasible anymore. It may also be necessary to re-calibrate every time the hardware changes. A framework that automatically learns the break-even points is necessary. We explain one approach for such a framework in Chapter 6.

# 4. Integrating Static GPU Tasks Into a DBMS

Researcher, companies as well as the open source community have already identified a wide range of tasks that can be solved on the GPU with benefit. In Section 3.3 we listed some libraries that are able to solve problems in different domains. In the old tradition of picking the low-hanging fruits first, the challenge is now to identify tasks in a DBMS that can benefit from these already available implementations.

As a first step we use a complete implementation for the K-Means algorithm on the GPU and show how this can be integrated into the system. K-Means is an example for application logic that is usually implemented by the DBMS-user and executed on top of the DBMS. Algorithms like this are not an integral part of the system but only use it to access the data stored in it. To the DBMS they are applications. These applications usually use SQL to extract the data once and then stop interacting with the system. However, the DBMS itself is often capable of executing this logic and users gain a number of advantages if they do so. In Section 4.1 we show, what the benefits of executing application logic in the DBMS are, how users can do this, and even use the GPU for such an approach.

Computational intensive algorithms can of course also be found in the internals of the DBMS. The query optimizer is one of the most complex components, which uses algebraic and numeric algorithms to find the fastest way to execute a query based on statistics. To do this, it has to solve algebraic equations with the help of matrix operations. In Section 4.2 we show, how we can use the *CUBLAS* library to enhance the optimizer for better estimations with the help of the GPU. Instead of using a given implementation—like in the section before—we combine the primitives of CUBLAS to port a previously CPU-optimized algorithm to the GPU.

Another internal task is the Delta Merge, which is specific to SAP HANA and some other column store DBMS as described in Section 2.2.4. In Section 4.3 we focus on the first part of this process: the creation of the new dictionary. There is no satisfying GPU-implementation for this problem yet. Hence, we have to develop an architecture-aware algorithm from the ground up. In the end our results show that the CPU beats the GPU easily because of the transfer bottleneck. Nevertheless, we can show that our algorithm is faster than combining primitives of the *thrust* library to achieve the same results. The dictionary merge is an excellent case study of how to port a simple inherently sequential algorithm to the GPU's architecture.

The three section explaining our tasks have the same structure: first, we explain the task and where it is used in a DBMS, then we provide details about the implementation and show the results of our evaluation. Before we conclude this chapter we discuss work

that is related to the presented tasks in Section 4.4.

## 4.1. GPU Utilization with Application Logic

The most use-cases for Co-Processors in a DBMS can be found in the application logic. While relational database operators are mostly data-intensive, compute-intensive algorithms, e.g., used for predictive analysis, or from the field of machine learning, do mostly just use the DBMS as data source and not as computing engine. The reason is that these algorithms often cannot be expressed (efficiently) with SQL.

Therefore, applications usually just extract data from the database and process it in-memory. By doing this, the application loses all the benefits a DBMS provides, e.g., internal indexes, which might be useful for the algorithm cannot be accessed. Also, the result of the application's calculations is not available to the DBMS directly. In consequence, if relational operators are needed during or after the special application logic was applied, the results have to be written to temporary tables or, and this is mostly the case and also the worse solution, these operators are implemented by the application. While the DBMS is specialized on these operators and therefore provides efficient implementations, most applications use naive approaches and lose performance and stability. In business logic we can often find nested loop joins for instance, which are implemented in scripting languages and therefore achieve execution times in the order of magnitude worse than DBMS operators. Another disadvantage of application logic on top is that the database cannot be modified directly. Instead the application always needs to map its copy to the database's original.

To overcome these disadvantages, most DBMSs provide a way to use application logic inside SQL statements, e.g., with the help of stored procedures. In this chapter we discuss how we can use the GPU for application logic in a DBMS and if we can benefit from that. Although we focus on MMDBMS in this thesis, we use DB2 for this experiment. To the best of our knowledge, DB2 has the most versatile support for external functions as we explain in the next section. SAP HANA will have similar support, called the *Application Function Library*, but the necessary functionality was not available in time. The K-Means implementation we use loads data into main memory before doing any calculation, which would not be necessary in an MMDBMS. During calculation there are no disk-operations, so we expect the results to be similar.

### 4.1.1. External Functions in IBM DB2

Co-processors can usually only be used with a low-level programming language such as C. For CUDA and Intel's Xeon Phi this code has to be compiled with a custom compiler. In OpenCL the compilation of the code that runs on the device is handled by a system library. The code responsible for calling the kernels can therefore be compiled with any C-compiler—or even in another language—as long as the binary can be linked to the OpenCL library.

Application logic written in vendor specific languages that are interpreted or compiled at run-time by the DBMS itself are therefore not able to use co-processors unless the

DBMS vendor explicitly implemented an interface. However, some DBMS, e.g., IBM DB2, are able to execute external functions—written by the user in C, Java and others—provided by a shared library, i.e., a DLL on Windows and an *.so*-file on Linux. In DB2 the user has to create a User-Defined Function (UDF) that tells the system where to find the shared library and maps the external function and its parameters to a method that can be called from a SQL statement. It does not matter how the shared library was created, as long as DB2 can find the specified function with the right interface in the binary file. The external function can either return a table—and therefore be used exactly like one in SQL statements—or it returns a scalar value such as the built-in *sum* or other aggregation functions. UDFs accept custom parameters at call time and can execute SQL statements to access the database.

Therefore UDFs are as flexible as if you would build applications on top of the database—with protocols like Open Database Connectivity (ODBC) to access data—and can be used in SQL statements as if their results were static data. They also allow developers to use any third-party library or code in the application. In many cases you can just use a ready application and replace the input and output logic as shown in the next section. The ability to use these functions from SQL statements can be a huge performance benefit because the DBMS can optimize with the knowledge about input data and the function. It can for instance cache the result of functions and re-use it in other statements. It may be able to stream and even parallelize processing by partitioning the data without any interaction of the application developer [15].

### 4.1.2. K-Means as UDF on the GPU

K-Means is a clustering algorithm that partitions data into $k$ clusters, in which each record belongs to the cluster with the nearest mean [65]. Although Lloyd did not use the name "K-Means", his algorithm is usually used for the clustering [95]. Given a set of coordinates with $n$ dimensions, his approach works as follows:

1. initialize $k$ means $m_1^1$ ... $m_k^1$ randomly, e.g., by picking the first $k$ elements from the input set.

2. find the "nearest" mean for every coordinate $x_p$: calculate the Euclidean distance:

$$S_i^t = \left\{ x_p : \forall 1 \leq j \leq k : \|x_p - m_i^t\|^2 \leq \|x_p - m_j^t\|^2 \right\}$$

   every $x_p$ is assigned to exactly one of the $k$ cluster sets $S_i$.

3. the means are moved to the centers of every cluster:

$$m_i^{t+1} = \frac{1}{|S_i^t|} \sum_{x_p \epsilon S_i^t} x_p$$

Steps 2 and 3 are repeated until no resp. a low percentage of the coordinates $x_i$ is assigned to another cluster.

### 4.1.3. Implementation

The purpose of our experiment is to use an available implementation for Lloyds's algorithm that we can execute in DB2 without the need for major modifications. Therefore, we used the open-source code developed by Giuroiu and Liao[1] with their example of clustering colors consisting of nine coordinates. The for us interesting part is how to built a UDF with this code.

Instead of reading the input from a file, we want to get it from a table and the cluster means should be returned as a table as well. So we implement our K-Means as external table function, the definition is shown in Listing 4.1. The function parameters are the number of clusters ($k$), a device identifier (CPU or GPU) and the name of the table with the input coordinates. It returns a table with the coordinates of every cluster mean. *EXTERNAL NAME* tells DB2 the name of the actual c function. The function itself does always return the same result on the same input data: it is *DETERMINISTIC* and does not access anything outside of DB2 (*NO EXTERNAL ACTION*). *NOT FENCED* means that it will be executed in the DB2 process itself. This is performance critical but also dangerous, because if the function failed unexpectedly, it would crash the DBMS as well. None of the parameters of our function can be NULL: *NOT NULL CALL* makes sure that DB2 never calls the UDF with a NULL parameter. *LANGUAGE C*, *PARAMETER STYLE DB2SQL* and *DBINFO* describe the interface for the function. For UDFs with the keyword *SCRATCHPAD* DB2 provides a memory segment with the size of 100 bytes, that is preserved between function calls. We get to the meaning of that when we describe the code. DB2 is able to split up the input in partitions and call the function once for every partition in parallel. We need to prevent this with *DISALLOW PARALLEL* because the parallel K-Means-logic is executed inside of the function. Other possible parameters for UDFs are described in [15]. The KMEANSCOLORUDF-function can be called with `select * from KMEANSCOLORUDF(2,'CPU','COLORS')` as soon as we have the shared library in place.

Listing 4.1: The definition of the K-Means UDF

```
1  CREATE FUNCTION KMEANSCOLORUDF(NUMCLUSTERS SMALLINT, DEVICE CHAR
       (3), TABLE_NAME VARCHAR(255))
2  RETURNS TABLE(C1 DOUBLE, C2 DOUBLE, C3 DOUBLE, C4 DOUBLE, C5
       DOUBLE, C6 DOUBLE, C7 DOUBLE, C8 DOUBLE, C9 DOUBLE)
3  EXTERNAL NAME 'cudaudfsrv!kmeansColorUDF'
4  DETERMINISTIC
5  NO EXTERNAL ACTION
6  NOT FENCED
7  NOT NULL CALL
8  LANGUAGE C
9  PARAMETER STYLE DB2SQL
10 SCRATCHPAD
11 DISALLOW PARALLEL
12 DBINFO
```

[1] available at https://github.com/serban/kmeans

Table functions like this are design in an Open-Next-Close (ONC) fashion, i.e., they are called once for every row they return. With this behavior they can be used like other relational operators in the Volcano model [36]. We get back to the details of this model in Chapter 5. One of the parameters given to the external function by DB2 marks which of the three phases this call belongs to. The scratchpad we mentioned earlier can be used to store pointers to allocated memory between calls. In case of the K-Means function we do all the calculations on the *open*-call and store the result to return it row-wise on every *next*-call. The work cannot be split between different calls because the first result is only available when the K-Means algorithm completed. The *close*- or *final*-call is used to free the memory used for the result.

Listing 4.2 shows the code for the open call. The original implementation by Giuroiu and Liao reads its input from Comma Separated Values (CSV)- or binary files into main memory. Keeping the data in main memory makes sense, because it is accessed multiple times during the calculation. Therefore, we also copy the data from a database table to a main memory segment. To read the data from the table as shown we can use a SQL from inside the UDF. DB2 supplies a custom pre-processor to provide convenient macros, such as *EXEC SQL PPREPARE*, for SQL execution. These macros are replaced with actual C-code before the compilation of the shared library.

In lines 1–6 we first query the number of records from the table to allocate memory (line 9/10) before the actual data extraction. The coordinates are then fetched and copied to main memory row by row in lines 12–27. In line 29 we allocate memory for the result and store the pointer in the scratchpad *pScratArea*. The unmodified code of the K-Means implementation is called in line 31 for the GPU or line 36 for the CPU. The threshold parameter responsible for the number of iterations in K-Means is fix in our implementation but could also be a parameter of the UDF. In lines 41 and 42 the input data is deleted from main memory because it is not needed anymore. The pointer to the result-row, the only data that has to be stored in the scratchpad, because it is needed for the next calls, is set to 0 in line 44.

Listing 4.2: Code for the open call (stripped error checking)

```
1  sprintf(strSmt,"SELECT COUNT(*) AS count FROM %s",table);
2  EXEC SQL PREPARE stmt10 FROM :strSmt;
3  EXEC SQL DECLARE c40 CURSOR FOR stmt10;
4  EXEC SQL OPEN c40;
5  EXEC SQL FETCH c40 INTO :count :countInd;
6  EXEC SQL CLOSE c40;
7
8  pScratArea->numObjs = count;
9  objects   = (float**)malloc(pScratArea->numObjs * sizeof(float*));
10 objects[0]= (float*) malloc(pScratArea->numObjs * pScratArea->
       numCoords * sizeof(float));
11
12 sprintf(strSmt,"SELECT C1,C2,C3,C4,C5,C6,C7,C8,C9 FROM %s",table);
```

```
13 EXEC SQL PREPARE stmt20 FROM :strSmt;
14 EXEC SQL DECLARE c50 CURSOR FOR stmt20;
15 EXEC SQL OPEN c50;
16 EXEC SQL FETCH c50 INTO :c1 :c1Ind, :c2 :c2Ind, :c3 :c3Ind, :c4 :
       c4Ind, :c5 :c5Ind, :c6 :c6Ind, :c7 :c7Ind, :c8 :c8Ind, :c9 :
       c9Ind;
17 objects[0][0]=c1; objects[0][1]=c2; objects[0][2]=c3;
18 objects[0][3]=c4; objects[0][4]=c5; objects[0][5]=c6;
19 objects[0][6]=c7; objects[0][7]=c8; objects[0][8]=c9;
20 for(int i=1;i<pScratArea->numObjs;++i) {
21   objects[i] = objects[i-1] + (pScratArea->numCoords);
22   EXEC SQL FETCH c50 INTO :c1 :c1Ind, :c2 :c2Ind, :c3 :c3Ind, :c4
         :c4Ind, :c5 :c5Ind, :c6 :c6Ind, :c7 :c7Ind, :c8 :c8Ind, :c9 :
         c9Ind;
23   objects[i][0]=c1;objects[i][1]=c2;objects[i][2]=c3;
24   objects[i][3]=c4;objects[i][4]=c5;objects[i][5]=c6;
25   objects[i][6]=c7;objects[i][7]=c8;objects[i][8]=c9;
26 }
27 EXEC SQL CLOSE c50;
28
29 pScratArea->membership = (int*) malloc(pScratArea->numObjs *
       sizeof(int));
30 if(strcmp(device,"GPU") == 0) {
31   pScratArea->clusters = cuda_kmeans(objects,
32     pScratArea->numCoords, pScratArea->numObjs,
33     pScratArea->numClusters, threshold,
34     pScratArea->membership, &loop_iterations);
35 } else {
36   pScratArea->clusters = seq_kmeans(objects,
37     pScratArea->numCoords, pScratArea->numObjs,
38     pScratArea->numClusters, threshold,
39     pScratArea->membership, &loop_iterations);
40 }
41 free(objects[0]);
42 free(objects);
43
44 pScratArea->result_pos = 0;
```

There is no actual application logic in the *next* and *close* of our UDF as shown in Listing 4.3. In every next call, we first check if DB2 already fetched all cluster means (line 2). If this is the case, we set the *SQLUDF_STATE* to inform DB2 that there are no more result rows. Else we copy the data to DB2's result row (7–15) and increment the result position (17). The *close*-call just frees the temporarily allocated memory (20-23).

Listing 4.3: Code for the next and close call (stripped error checking)

```
1 case SQLUDF_TF_FETCH:
2   if (pScratArea->result_pos >= pScratArea->numClusters)
3   {
```

```
4       strcpy(SQLUDF_STATE, "02000");
5       break;
6     }
7     i = pScratArea->result_pos;
8     *o_C1 = (double)pScratArea->clusters[i][0];
9     *o_C2 = (double)pScratArea->clusters[i][1];
10    *o_C3 = (double)pScratArea->clusters[i][2];
11    *o_C4 = (double)pScratArea->clusters[i][3];
12    *o_C5 = (double)pScratArea->clusters[i][4];
13    *o_C6 = (double)pScratArea->clusters[i][5];
14    *o_C7 = (double)pScratArea->clusters[i][6];
15    *o_C8 = (double)pScratArea->clusters[i][7];
16    *o_C9 = (double)pScratArea->clusters[i][8];
17
18    pScratArea->result_pos++;
19    strcpy(SQLUDF_STATE, "00000");
20    break;
21  case SQLUDF_TF_CLOSE:
22      free(pScratArea->membership);
23      free(pScratArea->clusters[0]);
24      free(pScratArea->clusters);
25    break;
```

Some additional code is needed to free memory and give an error message in case something goes wrong. The complete code can be found online under https://github.com/hannesrauhe/db2_cuda_udf.

### 4.1.4. Evaluation

We compare the UDF with the original implementation that reads from CSV files and outputs to the console, both on CPU and GPU. All measurements are conducted on the Z600 equipped with a Tesla C2050. [2] In Figure 4.1 we compare the total run-time of the four implementations for different $k$ and input sizes. $K$ has an influence on the number of iterations, because coordinates are more often assigned to other clusters. Additionally, more euclidean distances have to be calculated in step 2 of the algorithm if more cluster means are given.

For $k = 2$ the problem is data-intensive and the two clusters can be calculated in less than 100 ms with the original implementation. There is almost no difference between CPU and GPU, because the time needed for I/O dominates the execution time. It is surprising that the UDF is much slower in accessing the data than the file-based operation. Based on our experiment we find that reading 100 rows in DB2 takes approximately 1 ms longer than reading it as CSV from disk. This may be a result of fetching row by row with SQL or with the way we have to assign the read coordinates to the variables in the UDF.

For $k > 2$ I/O becomes less and less dominant, since more time is spent on the

[2] See Appendix A.1 for Details on Hardware.

actual calculation. Independent of the implementation we can see that the execution time not necessarily gets longer with a greater input size. With $k = 8$ and $k = 32$ for instance the data set with 45 000 rows is clustered faster than the larger set with 50 000 coordinates. This happens because the number of iterations—and the execution time with it—strongly depends on coordinates itself. Every data set for a fixed input size is randomly generated and therefore not related to the other sets.

With $k = 8$ we can see that I/O is still dominating, but also that the GPU is significantly faster in both implementations. While the pure computation time on the GPU is almost the same as with a smaller $k$, the CPU takes already three times longer. This becomes even more obvious with $k = 32$. The GPU is significantly faster than the CPU with the original implementation as well as with the UDF. With $k = 128$ the GPU implementation is only about 4 times slower than in the $k = 2$ version, while the CPU takes 70 times as long for the largest set. In relation the data input method plays no role anymore for this compute-intensive task.

### 4.1.5. Conclusion

In this section we have shown how we can integrate application logic that is executed by the DBMS itself. This has a number of advantages, compared to applications that use files for I/O or DBMS as a simple data source/sink. While file operations can be simple and fast, they lack flexibility when filter logic has to be applied or different input sources or output sinks are necessary. This extended I/O might be more complicated than the actual application logic. By using a DBMS as data source, application developers can use relational operators to pre-filter data before they apply the actual logic. Yet, they cannot use these operators on their intermediate or final results, because the data is not accessible to the DBMS. Here, the usage of UDFs allows a full integration of application logic into the database. The application becomes a relational operator itself and can be used in execution plans with all the usual benefits.

We have also shown that the effort to implement already existing application logic into UDFs is minimal and comparable to providing the logic needed to read and parse values from files. An implementation that runs on top of a DBMS requires similar effort to open and manage the DBMS connection.

UDFs are almost as flexible as stand-alone applications. For DB2 only some programming languages are supported, but since these are the popular ones, this will usually not be the limitation. Technically it is no problem to provide support for other languages as well, as long as it is possible to build a shared library. It was no problem to use CUDA in our shared library. Still, there are some limitations when it is necessary to use other shared libraries in the implementation of the UDF. In general they must either be loaded by DB2 (we did not find a solution for that) or they must be statically linked. We had no success in doing this with OpenMP; this is explicitly not recommended by the GCC developers; so we could not evaluate the parallel CPU implementation in our experiments. However, these problems are the exception, the general limitations are minor. DB2 or other systems might even be capable of lifting these all together in the future.

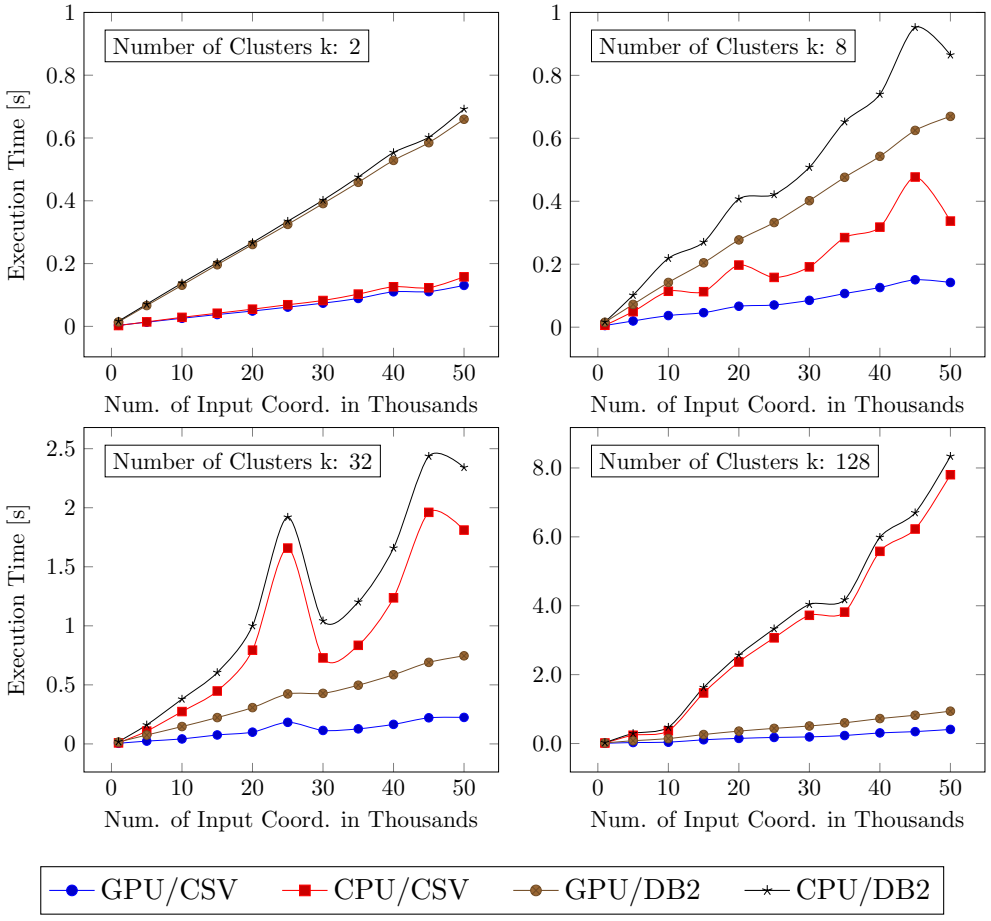



Figure 4.1.: K-Means on CPU and GPU

As we have seen, the performance of I/O operations becomes less important, the more computations are necessary on the data. In our experiment the file operations were faster than the DBMS, but this might change especially with MMDBMS or if different access patterns are necessary. The important discovery is that there is no difference for the run-time of actual calculations. We could show that we can benefit from the usage of GPUs in UDFs for compute-intensive calculations. The low effort for implementation combined with the advantages of having application logic in form of a relational operator clearly outweighs the minor performance loss for compute-intensive tasks.

## 4.2. GPU-assisted Query Optimization

In the last section we discussed K-Means as representative task triggered by the DBMS user. In this section we focus on compute-intensive tasks that are used by the system for its everyday operations, e.g., in the query optimizer. Before a query is executed, the DBMS creates an execution plan and chooses the order and type of operators for fast execution. The optimization is based on statistical methods and heuristics and can get very compute-intensive. Of course the optimization should be done in a fraction of the time needed to execute the user's query itself. Because of the limited time span, not all methods to enhance the plan are used, although they might lead to better execution plans. By using the GPU's raw execution power to apply more complex optimization methods in a similar time span, we can achieve better query execution times. We describe the problem of selectivity estimations and introduce the Maximum Entropy (ME) approach proposed by Markl et al. [67]. The Newton method we use to solve the ME problem is not only faster on the CPU than Markl's original implementation but can also be executed on the GPU. Especially when handling large matrices the GPU is significantly faster than the CPU as our evaluation shows.

### 4.2.1. Selectivity Estimations and Join Paths

Since SQL is a declarative language, there are a number of different ways to actually execute a query in a RDBMS. The query optimizer is responsible for finding the optimal access path for a query [38]. Based on knowledge about the data stored in the database it decides about the type and order of relational operators to execute. A simple example for this is a select statement with two conditions $p_1$ and $p_2$. If the selectivity $s_1$ for $p_1$ is higher than $s_2$, i.e., more rows are returned if $p_1$ is the only condition than if $p_2$ is the only condition of the statement, the optimal plan would evaluate $p_2$ first. This way the second operator has less values to check and can finish faster. In this simple example only the selectivities of single predicates are needed. If the query requires a join of a relation with the result of the above selection, the optimizer chooses the join operator based on the size estimation of this result. Therefore, we need to know the combined selectivity $s_{1,2}$for $p_1$ and $p_2$.

While the selectivity of a predicate on a single attribute can be determined fairly accurately, e.g., by relying on histograms, doing so for conjunctive predicates involving the joint frequency distribution of multiple attributes is much harder. Usually, we

assume that columns are independent from each other and calculate the combined selectivity $s_1..s_n$ of n predicates $p_1..p_n$ by multiplying their selectivities. However, often this so-called independency assumption is not correct: a very common example in research literature is a table that stores car models and car makers among other attributes. Obviously the independency assumption would lead to wrong estimations and non-optimal execution plans, because usually a model is only made by one car maker. The same is valid for tables storing cities, zip codes and countries. To overcome the problem, some DBMS store Multivariant Statistics (MVS). They are approximated for a subset of combinations with the help of multi-dimensional histograms [84] for instance. However, it is not feasible to calculate, store, and update these joint selectivities for every combination of columns. Instead the available knowledge has to be combined if a query requires a joint selectivity not stored.

The problem of calculating estimations when joint selectivities are available is complicated, because there are different ways of combining available selectivities. Let's assume, that we need the selectivity $s_{1,2,3}$ and have MVS for $s_{1,2}$ and $s_{2,3}$ as well as all single selectivities. The needed selectivity can be calculated in two ways: $s^a_{1,2,3} = s_{1,2} * s_3$ or $s^b_{1,2,3} = s_{1,3} * s_2$. It is likely that $s^a_{1,2,3} \neq s^b_{1,2,3}$. We cannot predict which of the two estimations is better. Therefore, traditional optimizers decide for one of the two choices[3] and ignore available knowledge. To ensure that every plan is costed consistently the optimizer also has to remember each choice and make sure that it does not use another choice for similar plans. This approach therefore requires additional book-keeping. Even worse, since once discarded MVS are also discarded later, the optimizer tends to use the plans about which it know the least [67]. A method to use all available knowledge would certainly produce better estimations and ensure consistent plans. The *ME* approach for selectivity estimations proposed by Markl et al [67] is such a method, but it also requires calculation power.

### 4.2.2. Maximum Entropy for Selectivity Estimations

To estimate the selectivities we consider all available knowledge. The principle of ME assumes that the best estimation is the one with the largest entropy [40]. Entropy is a measure of uncertainty for a probability distribution $q = (q_1, q_2, ...)$ defined as

$H(q) = -\sum_i q_i \log q_i$

For Markl's approach we try to maximize the entropy function $H(q)$ while keeping it consistent with the knowledge about certain selectivities.

Without any knowledge there is only one single constraint that the sum of all probabilities equals 1. The ME principle assumes a uniform distribution. If only single column selectivities are use, the ME principle is similar to the independency assumption. However, it allows us to also consider MVS as well.

To build the system of linear equations all possible combinations of a given set of predicates $P = (p_1, p_2, ..., p_n)$ need to be considered. A combination is called *atom* and denoted as binary string $b$ of length $n$ for a term in disjunctive normal form, e.g., one

[3] Of course, there is also the third way to simply multiply the three single selectivities.

atom for $n = 2$ is: $p_1 \wedge \neg p_2$, $b = 10$. The selectivity for this atom is denoted as $x_{10}$. Each predicate $p_i$ is influenced by one or more atoms. The set $C(i)$ denotes all contributing atoms. In the example above $x_{10}$ and $x_{11}$ contribute to $p_1$: $C(1) = \{10, 11\}$. If a selectivity $s_i$ is known, $i$ is part of the knowledge set $T$.

The selectivities $s_j$ for $j \notin T$ are calculated according to the ME principle by solving the *constrained optimization problem*:

$$\min_{x_b | b \in \{0,1\}^n} \sum_{b \in \{0,1\}^n} x_b \log x_b$$

constrained by the known selectivities in $T$:

$$\sum_{b \in C(i)} x_b = s_i, i \in T$$

In the example above $s_1$ and therefore the following constraints are given:

(I) $s_1 = x_{10} + x_{11}$
(II) $s_\emptyset = \sum_{bin\{0,1\}^n} x_b = 1$

$S_\emptyset$ is always known. In general a numerical method is needed to solve the problem. Because of the consideration of all possible combinations, the time needed to compute the solution grows exponentially with the number of predicates $n$. Markl et al. proposed the iterative scaling algorithm. In the following section we present the *Newton approach*, which is also suitable for execution on the GPU.

### 4.2.3. Implementation of the the Newton Method

Listing 4.4: "The Newton approach"

```
1  function [u v] = boltzmann_shannon_newton(A,d,abort_cond);
2  %m < n
3  [m,n] = size(A);
4  v = zeros(m,1); u = zeros(n,1); Aty = zeros(m,1)
5  criterion = abort_cond + 1;
6  eb = exp(1)*d;
7
8  tic
9  while criterion > abort_cond
10   u_old = u;
11
12   eAty = exp(-Aty)
13   rs = eb - A*eAty
14   AA = A*diag(eAty)*A';
15   dd = AA\rs;
16   v = v - dd;
17
18   criterion = A'*v;
19   u = exp(-1 - criterion);
20   criterion = norm(u-u_old,inf);
```

```
21
22    Aty = A'*v
23    j = j+1;
24  end
25  toc
26  j
```

The Newton approach shown in Listing 4.4 iteratively calculates an approximation for the selection based on the values of the last iteration. In matrix $A$ there is a column for every predicate and a row for every possible combination of these predicates, i.e., $A$ has a size of $m \times 2^m$. $d$ contains the known selectivities. The shown code estimates the selectivities $u$ for all possible combinations of predicates with an iterative approach. The iteration stops as soon as the difference between the previous and this iteration's result gets smaller than the specified accuracy *abort_cond*. Because an iteration depends on the previous one a parallel execution of the outer loop is not possible.

Therefore, we can only use a parallel implementation for the algebraic functions to calculate the intermediate vectors and matrices. However, since the matrix grows exponentially with the number of predicates, the computation easily justifies the overhead. No calculation is done on the CPU since this would require to transfer the large matrix. Instead, we generate the matrix on the GPU, i.e., only the known selectivities need to be transferred. Afterwards all computations involving the matrices, even the ones that would run faster on the CPU, are executed on the GPU to avoid transfer of intermediate results. However, every matrix operation requires a full synchronization—calling a new kernel—from the CPU. Hence, the loop runs on the CPU, as well as check of the abort condition.[4] For the actual matrix operations the CUBLAS library is used (cf. Section 3.3).

### 4.2.4. Evaluation

Markl et al. already proved that queries benefit from the better estimations [67]. Here we show that we achieve the same quality faster. In Figure 4.2 the execution times for the different approaches on our K20 machine are shown.[5] The results of Markl's algorithms are marked as *Original*, our Newton approach is shown in three variants: the sequential implementation, which uses one CPU core, the Multi-Core implementation (*N. MC)*), and the GPU implementation (*N. GPU)*. Both plots show the same numbers, the left-hand one on a logarithmic scale, the right-hand one on a linear scale for the important sub-second time-span. Longer running estimations are impractical, because then the query optimization time may take longer than the actual query execution time. However, this is only a rule of thumb. Surely, there are scenarios where it makes sense to wait longer for a good estimation as well as scenarios, where a second is also much longer than the actual execution.

[4] In a second implementation we use Dynamic Parallelism and check the abort condition on the GPU as well. Similar to our results in Section 3.4.1 the performance is worse.

[5] See Appendix A.1 for Details on Hardware.

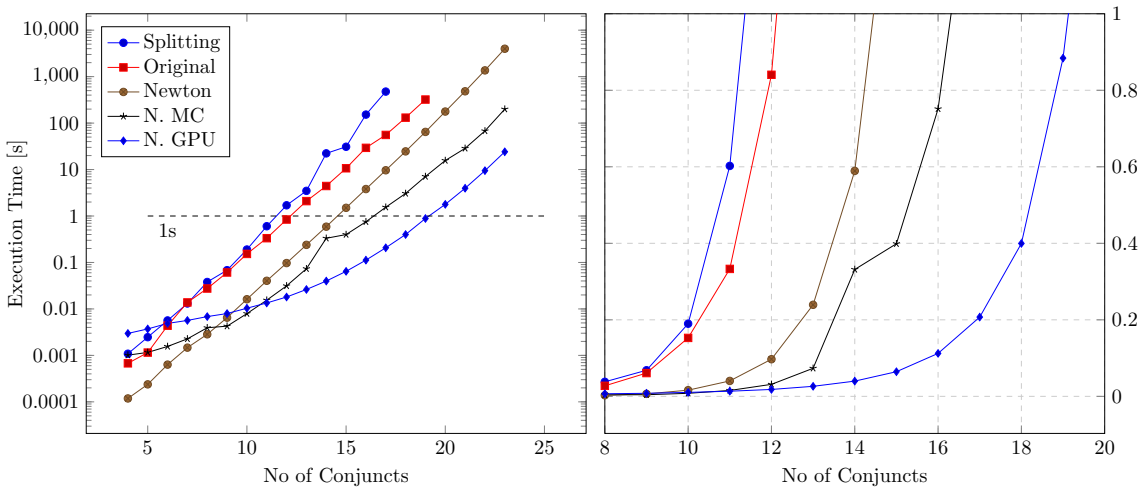



Figure 4.2.: Comparison of run times of the maxentropy algorithms

In Figure 4.2 we see that the original approach determines an estimation within a second for up to 12 predicates on our hardware. The Newton approach is 10 times faster than the original. It calculates an estimation on up to 14 predicates, the OpenMP implementation on 16, and the GPU implementation even on 19 predicates; all within one second. Within 10 ms we can estimate selectivities for up to 10 conjuncts with the help of our Newton-approach. The break even-points are at 8 conjuncts, where the parallel CPU implementation is faster than the sequential one, and 11, where the GPU is the fastest of all implementations. There is a considerable variance in the performance of the original and the splitting algorithm of up to 50% depending on the data. The variance for the parallel CPU implementation is also very high, which might be due to the partitioning of the OpenMP framework. The sequential Newton algorithm as well as the GPU implementation deliver stable run times, which vary around 10–15%.

Overall we can conclude that the GPU is well suited for this type of workload. Because of the low amount of data to be transferred, the PCIe bus is not the bottleneck. The amount of of memory on the external card is also no problem for estimations up to 23 predicates. At this point the CPU takes more than 10 min to calculate. The ME approach seems to be not an option for this size. The GPU delivers the most stable run times and speedups up to 100 times compared to one core on the CPU and up to 10 compared to 16 cores. In contrast to the query execution experiments discussed in Chapter 5, the break-even point is easy to find for an optimizer. As long as the hardware does not change, it can easily decide whether to use GPU or CPU for the estimation based on the number of predicates.

### 4.2.5. Conclusion

In this section we have shown that we can benefit from GPUs by using them for the complex calculations executed by the query optimizer. Using the functions of the *CUBLAS*-

library requires only minimal knowledge about the GPU's architecture. Markl et al. have already shown that it is worth to use the ME approach for query optimization, because usually there is less time needed for calculations than we save during execution of the query. However, the more predicates are involved in the query the longer the optimization takes; at some point it is better to execute a query plan based on less accurate estimations. Markl states that one second is the upper limit for optimization time. With the GPU we were able to estimate selectivities for up to 19 predicates within one seconds compared to 14 on the CPU. At this point we are 80 times faster than the same approach on a single thread on the CPU and 300 times faster than the original method. The GPU also out-runs all cores on our machine by a factor of 10. Additionally, the CPU is free for other operations during selectivity estimation.

## 4.3. The Dictionary Merge on the GPU: Merging Two Sorted Lists

The Delta Merge process in HANA is necessary to integrate changes from the delta buffer into the main storage of the database. It has to be executed regularly and therefore sets systems with a high number of tables under frequent load. If it was possible to offload this algorithm to the GPU, this would free resources on the CPU and in consequence possibly speed up the whole system.

As we explained in Section 2.2.4, dictionary encoding is used in main as well as in delta storage. The creation of a common dictionary for the new merged main storage is therefore the first step of the merge. Both dictionaries can be accessed in an ascending order, hence, the new dictionary can be created by simply iterating over the two sorted structures as listed in Algorithm 1. This is a well-known problem, where parallel implementations for CPU and GPU are widely available. The GNU C++ library provides a parallel version of *std::merge*, which can be activated with a simple compiler switch [25]. CUDA's *Thrust library*—equivalent to the C++ standard library on NVIDIA GPUs—also provides a parallel implementation [48].

However, values may occur in both dictionaries. Therefore we need to modify the algorithm in a way that it eliminates these duplicates while merging. The modified pseudo code for the sequential algorithm is listed in Algorithm 2. We will refer to this algorithms as *Dictionary Merge*. A GPU implementation can be achieved by executing the *unique* primitive of the Thrust library after the *merge* primitive. Unfortunately, this means touching the data twice and synchronizing in between. Because performance is essential for this task, we decided to implement our own merge algorithm for the GPU that does merging and duplicate removal in one run.

### 4.3.1. Implementation

In Section 3.1.2 we explained that the GPU is a mixed architecture in the sense of Flynn's taxonomy. A parallel algorithm can only use such an architecture efficiently if it provides two levels of parallelism. On the case of the first part of the Delta Merge,

**Algorithm 1** Sequential merge algorithm

**Input:** $list_A, list_B$
**Output:** $list_C$

```
1: i_A = 0
2: i_B = 0
3: i_C = 0
4: while i_C < sizeof(list_A + list_C) do
5:    if list_A[i_A] <= list_B[i_B] then
6:       append(list_C, list_A[i_A])
7:       i_A = i_A + 1
8:    else
9:       append(list_C, list_B[i_B])
10:      i_B = i_B + 1
11:   end if
12:   i_C = i_C + 1
13: end while
```

the creation of the dictionary, we show how algorithms have to be designed to meet this requirement.

#### First layer of parallelism

The parallelization of the Dictionary Merge for CPUs is straight forward. It is obvious that the loop cannot be executed in parallel without an modification since every loop iteration depends on the previous one. However, we can use the divide and conquer approach in three phases: partition both lists, execute the sequential merge on every pair of partitions, and merge the partitioned results in the end. A small amount of calculations is necessary in the first phase to find corresponding partitions: we split the first list in chunks of equal size and do a binary search for the last element of every chunk to find the right partition boundaries of the second list. For the second phase we execute the sequential merge algorithm on every pair of partitions.

In general it makes sense to use as many threads as cores are available on the CPU. If less threads are used some cores are idle during calculation. This is called *under-subscription*. *Over-subscription*, i.e., using more threads than cores, is also not optimal, because then one core has to work on more than one thread. This would lead to context switches: cache and registers are flushed to load the other thread's from main memory. Depending on the Operating System (OS) scheduler this could happen multiple times per partition. However, if every core works on only on partition pair, there is a high chance of under-subscription, because threads that work on small partitions finish early, while large partitions are still processed. To balance the workload it is therefore necessary to use one thread for multiple partitions. As this is a general problem, frameworks such as Intel's Thread Building Blocks (TBB) introduce an abstraction layer. Instead of threads, the developer starts tasks. Theses tasks are assigned to the optimal number of threads,

**Algorithm 2** Dictionary Merge: merge algorithm with duplicate elimination

**Input:** $list_A, list_B$
**Output:** $list_C$

```
1:  i_A = 0
2:  i_B = 0
    Ensure, the greatest element is at the end of list_A:
3:  append(list_A, last(list_A) + last(list_B))
4:  while i_A < sizeof(list_A) do
5:    if list_A[i_A] < list_B[i_B] then
6:      append(list_C, list_A[i_A])
7:      i_A = i_A + 1
8:    else if list_A[i_A] > list_B[i_B] then
9:      append(list_C, list_B[i_B])
10:     i_B = i_B + 1
11:   else
12:     append(list_C, list_A[i_A])
13:     i_A = i_A + 1
14:     i_B = i_B + 1
15:   end if
16: end while
    Remove the element inserted in line 3:
17: pop(list_C)
```

solving the problem of under- and over-subscription as long as enough tasks are queued.

For the unmodified merge, it is not necessary to combine the merge-results of each partition, because their size is known in advance, so the result partitions can be written directly to the target memory. However, for the parallel version of the Dictionary Merge, we do not know the size of the result partition, because the number of duplicates is not known. Therefore, we have no choice but to create every result partition in a memory segment that is large enough to hold the number of elements in both lists, which is the maximum size in case there are no duplicates at all. Hence, the third phase is necessary to copy the results of each thread to contiguous memory segments. It is possible to do this in parallel as well, but we have to determine the right position for every intermediate result by calculating the prefix sum over all intermediate result sizes.

#### Second layer of parallelism: SIMD

There are two problems with the first layer approach on GPUs. First, to use every core on the GPU we would need approximately 1000 partitions. However, to fully utilize these cores at least 10 000 partitions would be necessary as shown in Chapter 5.5.4. Therefore, the overhead for finding the partition boundaries and copying the results in the end would get significant. Second, executing this algorithm in parallel does not favor the SIMD architecture nor the memory architecture of the GPU.

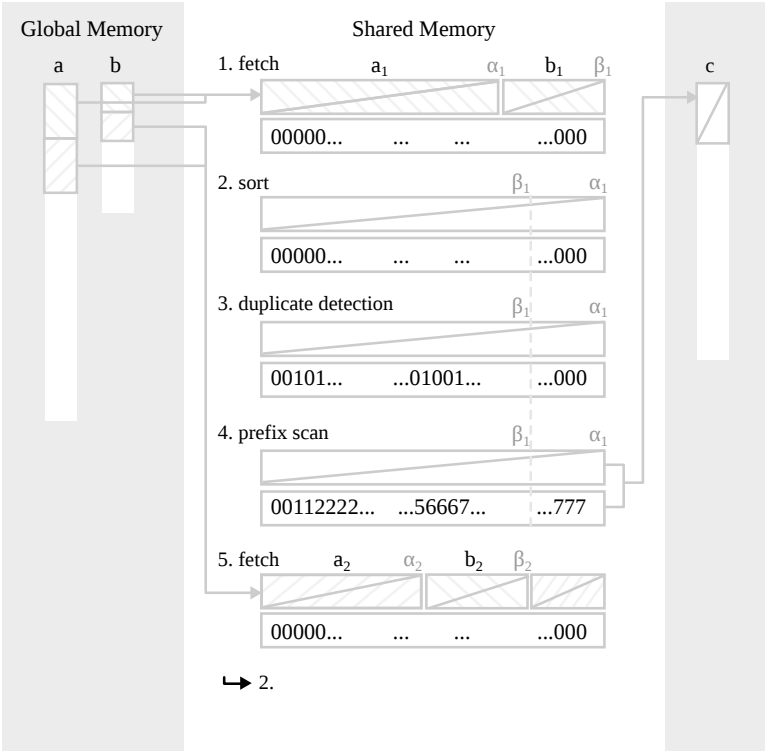



Figure 4.3.: The parallel zip merge algorithm

We propose an algorithm that can be executed in parallel in one workgroup and is adapted to the GPU's memory hierarchy. Instead of working element-wise like the sequential algorithm, we compare blocks of both lists as shown in Figure 4.3. Each step of our algorithm is vectorized and we minimize the access to device memory by copying blocks of both lists into the shared memory of the SMX in step 1. In step 2 we use the *bitonic sort* implementation of the CUDA SDK [74] to sort all elements in shared memory. In step 3 every element is compared with its right neighbor. We use another segment of shared memory to mark the position of every duplicate with a one. In step 4 the *prefix sum* implementation from the CUDA SDK is used to sum up all ones in this segment. At this point we know how many duplicates occur left of every element in the sorted list. With this knowledge we can copy the sorted list back to the global memory ($c$) and remove the duplicates in the process. The problem is, that one fetched block can overlap with the next block in the other list. After each *fetch* we have to compare the greatest elements in both lists $\alpha$ and $\beta$. If $\alpha > \beta$, all elements (from both lists) smaller than $\beta$ are in shared memory or have been processed earlier. As a consequence all elements greater than $\beta$ stay in shared memory in step 5 and because these elements must be from list $a$, we fetch lesser elements from this list for the next run. If $\alpha < \beta$ we fetch a smaller amount of elements from list $b$ instead. The threads have

to be synchronized after each step and shared memory is used to communicate between threads. This is cheap on the thread-level, but very expensive on the workgroup-level. Therefore, we use this algorithm only as second layer of parallelism, while we use the first layer to distribute work between workgroups. We vectorize execution inside workgroups and distribute the work in partitions over a higher number of these. We call this approach for the Dictionary Merge *Zip Merge*.

### 4.3.2. Evaluation

We are interested in two things for our evaluation. First, is the Zip Merge faster than the two combined primitives of the Thrust library? Second, can we beat the CPU with this approach? Our experimental setup is a possible merge scenario: $list_A$, the dictionary of the main storage, has five times as many elements as $list_B$, the delta dictionary, and half of the elements of $list_B$ also occur in $list_A$. In Figure 4.4 the kernel execution time of the two GPU implementations and the time needed for partitioning and merging on the CPU are shown; the Z600 with the Tesla C2050 was used for the experiment.[6] Our implementation performs between 20 % and 40 % better than the Thrust library, which is expected because it touches the data only once. However, it is also only 20 % to 40 % faster than a sequential CPU implementation.

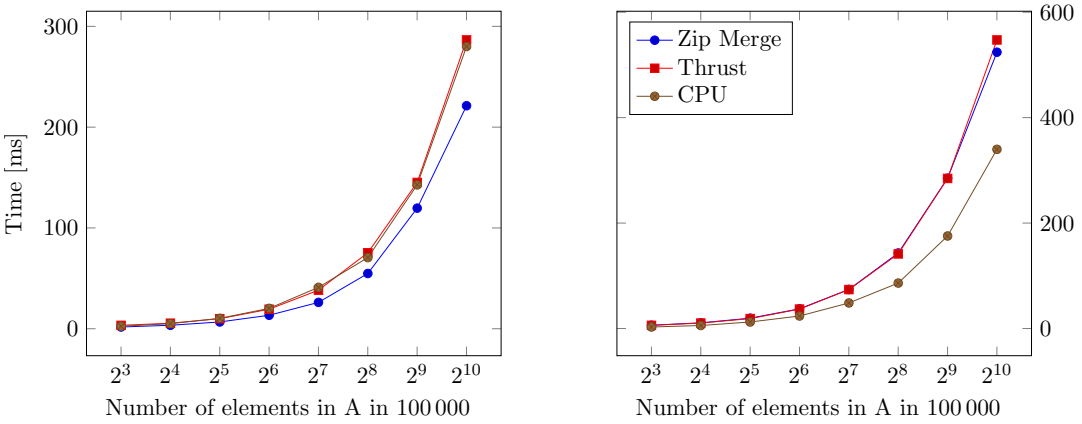



Figure 4.4.: Comparison of Thrust's merge/unique[75] and our implementation kernel execution time left, total execution time right

For the total measurement we included the final copy from the partition results to the target memory. For the sequential version this is necessary, because we had to allocate enough memory to save the largest possible result in case no duplicates occur in both dictionaries. The real size of the result is first known, when the process finished. Therefore we have to allocate the right amount of memory in the end and copy the result there. In a DBMS this may be handled by special allocators. On the GPU this is part of the PCIe transfer to main memory. The CPU implementation—using only one core—is

[6] See Appendix A.1 for Details on Hardware.

significantly faster than transferring the lists to the GPU and merging the lists there as the right-hand curve in Figure 4.4 shows. In contrast to the recoding algorithm in Section 3.4.3 the GPU does not scale better than the CPU with a growing input size.

Unfortunately, we cannot use streaming either because of the way we have to partition the lists in the beginning. The overhead needed to parallelize and vectorize the algorithm to fit onto the CUDA architecture is too high to compensate for the transfer.

### 4.3.3. Conclusion

In this section we were able to show that we cannot expect the GPU to speed-up all operations executed by the DBMS. Despite the effort we spent on porting the native dictionary merge algorithm we were only able to achieve little speed-up compared to a sequential CPU implementation. If we consider the necessary transfer times the GPU even is considerably slower than the CPU. There are a few reasons why this algorithm is not a good match for the GPU considering the criteria from Section 3.5. There is a working parallel version for the CPU that introduces negligible overhead for partitioning, so point 1 is fulfilled. However, as one can already guess from the complexity, the Zip Merge involves much more calculation than the simple Algorithm 2. The same is true for *thrust::merge*, but the duplicate elimination adds a few additional steps. After all, a bitonic sort and a prefix sum is needed for every partition instead of one comparison per item in the sequential version. Also, 80 % of the time needed to execute the algorithm on the CPU is needed to transfer input and results from/to the GPU, so point 3 is barely fulfilled. Finally, we cannot efficiently overlap transfer and computation because the complete dictionaries are needed for the partition step.

There is another major problem we have not mentioned yet: the Zip Merge does not work with variable length strings. There are several problems, most notably we do not know the amount of shared memory needed. It is limited and has to be allocated for a kernel in advance. We cannot predict the memory needed for every partition with variable length strings. Maybe it is not even possible to adapt this approach at all. Certainly, the computation would require more inter-thread communication and dynamic adjustment of partition sizes.

Hence, we can conclude that the GPU is not suited to handle the Dictionary Merge. However, the merged algorithm is better than the two combined primitives from the Thrust library. Additionally, the Zip Merge algorithm is a good showcase because there is a clear difference between the SIMD parallelism needed for work distribution inside a workgroup and the “independent” parallelism between workgroups.

## 4.4. Related Work

Application logic in HANA was already discussed in [39]. They integrated the R language for statics as basis for machine learning algorithms. SQLScript provides the ability to express procedural logic in general and is interpreted by HANA itself [9]. Another approach to execute K-Means with the help of the DBMS was presented by Ordonez

et al. in [81]. They implemented the whole algorithm in plain SQL with the help of temporary tables.

Heimel and Markl also identified query optimization as a promising task to execute on the GPU, because it is usually compute-bound and requires minimal data transfer. As a proof-of-concept they integrated an GPU-assisted selectivity estimator for real-valued range queries based on Kernel Density Estimation into PostgreSQL [45].

Krüger et al. showed that the GPU is better at merging than the CPU [61]. However, on the CPU they chose to fully sort the dictionaries after concatenating and did an additional step for eliminating duplicates instead of using Algorithm 2. Also, in the second phase of the Delta Merge, when the new value IDs are created (see Section 2.2.4 and Section 3.4.3), they did not use a simple map, but looked up every value in the new dictionary. Both decisions make the algorithm do much more computing than necessary and create an advantage for the GPU.

# 5. Query Execution on GPUs—A Dynamic Task

In the last chapter we have taken a look at static tasks in a DBMS that can be offloaded to the GPU. Their common ground is that we can predict the benefit of offloading if we know their input's size and their parameters, e.g., the $k$ for K-Means or the number of predicates for the maximum entropy calculations. We discuss this scheduling decision in detail in the next chapter. Still, to justify an expensive co-processor like a high-performance GPU in a general DBMS, it should be able to execute at least parts of the major workload: query execution.

In contrast to static tasks, query execution is nearly unpredictable in performance on the GPU, because it is a combination of different primitives—relational operators in this case—that are combined at run-time. The order and type of operators is different for every query. Hence, we call query execution a *dynamic task*. Past research on using GPUs [34, 46, 43, 42, 6, 54] and other co-processors [31, 97] for query execution treated every operator as a static task independent of the other operators. We present parts of this work in Section 5.6.

There are, however, problems with the approach of considering every operator on its own:

- First, although especially OLAP queries can be quite compute-intensive, their execution plans are often very large and most operators actually do not do much work. The overhead for starting new kernels and synchronization becomes significant and we cannot fully utilize all cores on the GPU. In Section 4.3 we have seen that we gain performance if we combine primitives. This would be beneficial for query execution as well.

- Second, we have to materialize every intermediate result for this approach and in the worst case transfer it to main memory if it does not fit into the device memory. Since the output of every operator is the input of the next one, we do not know their parameters and input sizes in advance. Hence, we cannot decide if offloading an operator is beneficial before we start the actual execution. Instead we have to wait for every operator to finish and decide where to execute the next one.

To overcome these problems, we proposed to build custom query kernels to avoid materialization and overhead in [88].[1] These kernels represent the query execution plan and can be compiled to machine code at run-time. JIT-compilation was recently proposed as a novel approach for query execution on the CPU as well. In Section 5.2 we explain the

[1] This chapter represents are more detailed description of our work in [88].

concept and why this approach is better suited for the GPU than the classic operator approach. Dees et al [19] introduced a bulk-synchronous model that merges relational operators in a way that most queries are transformed to not more than two function calls. However, in their work they mainly focus on data representation, indexes and the techniques used for execution of the (merged) operators. In Section 5.3 we go into the details of the bulk-synchronous processing they use for execution on multi-core CPUs.

Because their model uses only one layer of parallelism, it cannot be applied to the GPU in this form. By adding a second SIMD-stlye layer of parallelism—similar to what we explained for the Dictionary Merge above—we can adapt this model for GPU execution as we show in Section 5.4. We compare the performance of our prototype on GPU and CPU with the implementation from [19] in Section 5.5.

Still, in general query execution is a data-intensive task. Even with a solution that prevents back-and-forth-transfers of intermediate results, input and output of the whole query must be transferred between CPU and GPU somehow. Therefore we take a look at our options to cope with this problem in Section 5.1 before we start off.

## 5.1. In General: Using GPUs for data-intensive problems

The transfer to the GPU is the main bottleneck for data-centric workloads as we already discussed earlier. Because of two characteristics, query execution is data-intensive and not compute-intensive, making the transfer to the external co-processor an unsolvable problem.

First, we cannot predict beforehand which values of a relation are really accessed and have to transfer them all. If a selection evaluates a predicate on one column, the values of the other column in this row are also filtered without ever been touched. Universal Virtual Addressing (UVA) is NVIDIA's solution to avoid unnecessary transfers by streaming the required data at the moment it is needed on the GPU. The authors of [52] propose UVA for join processing on the GPU, Yuan et al. also use it for query execution in general [106]. But this approach is very limited because just like the "pinned transfers" described in Section 3.4.3 UVA requires main memory to be pinned. As we have shown, pinning memory is very expensive and only a limited amount of memory can be pinned. Also, we have no way to force data to stay on the GPU, so this approach might lead to repeating transfers of the same data segments if data is touched more than once. Still, this might be a solution for some problems.

The second and much worse problem is that the transfer of a tuple is slower than processing it once in typical query. Even if the GPU accesses the main memory sequentially and we can use streaming we remain limited to the PCIe bandwidth of theoretically 8 GB/s (PCIe 2). Research has shown, that evaluation of predicates is much faster—up to 25 GB/s per thread(!)—on the CPU [103]. Even if other operators are slower on the CPU, the probability that queries with a low selectivity are faster on the GPU is low. From our perspective there are three possibilities to cope with this problem:

1. The data is replicated to the GPU memory before query execution and cached there.

2. The data is stored exclusively in the GPU's memory.

3. CPU and GPU share the same memory.

The first option limits the amount of data to the size of the device memory, which is very small at the moment: high end graphic cards for general-purpose computations have about 4–16 GB. Furthermore, updates have to be propagated to the GPU.

Storing the data exclusively in the GPU's memory also limits the size of the database. Furthermore, there are queries that should be executed on the CPU, because they require advanced string operations or have a large result set (see Section 5.4.2). In this case we face the transfer problem again, only this time we have to copy from GPU to CPU.

The third solution, i.e., CPU and GPU sharing the same memory, is not yet available for high performance GPUs. Integrated GPUs in AMD's Trinity architecture already show a working solution for this feature, but both the CPU as well as the GPU are slow compared to modern server processors and graphic cards. However, research has already evaluated join operations on such architectures [54]. So, we are sure that the transfer bottleneck for GPGPU applications will not be a problem for long anymore. But until shared memory is available, replication is the best choice for our needs.

## 5.2. JIT Compilation—a New Approach suited for the GPU

Query execution in most RDBMS is done in two phases. In the first phase the DBMS parses the query entered by the user and creates an execution plan consisting of operators of a system-specific algebra. This plan is optimized during its creation—we explained some details in Section 4.2—before it is actually executed in the second phase. For the execution of this plan traditional RDBMS use the Volcano model [36]. Every operator of the execution plan has an ONC interface. After an initial `open`, `next` is called on the final operator to get the first record of the result. The operator then calls the `next` function of the parent operator and so on until the whole result set is fetched record by record. The model is flexible enough to allow the introduction of new operators, e.g., different join-implementations and by calling the operator for each tuple it avoids materialization of intermediate results. With this strategy the bottleneck of traditional systems, I/O to disk, can be reduced to a minimum. With larger main memory buffer pools in disk-based DBMS and MMDBMS, I/O is still dominating in some cases, but not the general bottleneck anymore. Hence, the overhead induced by the tuple-at-a-time-strategy is much too high for modern systems. The code has to be loaded by the CPU every time one operator's next method is called, the code locality of the strategy is bad. Additionally, these methods are virtual, i.e., every call requires a lookup in the virtual method table.

Consequently, MonetDB—as the pioneer column store DBMS—uses a column-at-a-time approach, which calculates the whole result of each operator in one call. This approach perfectly supports intra-operator-parallelism. Parallel execution of each operator can be easily implemented by splitting the input of the operator, processing it in parallel, and merging the results in the end. The disadvantage of this is that every

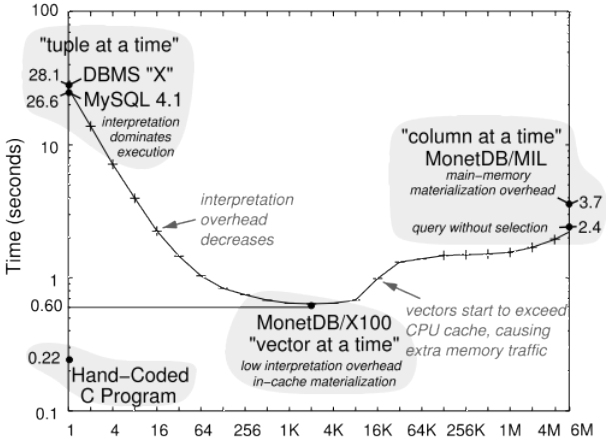



Figure 5.1.: Hand-written code is still faster than vectorwise [108]

operator runs until all data is processed and its results are materialized in main memory. Therefore, every operator needs to load the intermediate results into the cache again. Processor optimizations, such as branch prediction over operator bounds, are unusable. Another problem is that the size of intermediate results is limited to the available main memory. If it gets larger, the operating system starts to swap to disc.

To overcome this limitations and get the best of both approaches, the successors of MonetDB, X100 and Vectorwise use configurable-size vectors of tuples as intermediate results [108]. By choosing a size smaller than the available cache, materialization is very fast, but the workload of every next-call is still large enough to support intra-operator-parallelism. Also, the overhead for book-keeping and operator calls is dramatically minimized compared to the tuple-at-a-time model. Vectorwise shows impressive performance on OLAP scenarios with this vector-at-a-time processing [10].

Still, all of these approaches use the operators introduced by the Volcano model to separate each phase of the query-plan execution. Hand-written C-Code still performs significantly faster as we see in Figure 5.1, because there is no book-keeping and no materialization. Neumann [72] proposes to leave the volcano-like operators and merge them until synchronization is unavoidable. Logically, these operators are still used, but they are not called as separate function anymore. Figure 5.2 shows an example for this approach. He generates code for every query, compiles it just-in-time and uses the compiler for low-level-optimization. Neumann already stated that new compiler developments—like adjustments to modern CPUs—can directly be used with the help of the JIT approach.

The Volcano model with operators as functions performs even worse on GPUs, because kernel calls are much more expensive than virtual function calls. The column-at-a-time model requires intermediate results to be transferred to the CPU's memory because the GPU's memory might not be large enough to hold it. With the vector-at-a-time model we would have to choose huge vector sizes to allow full utilization on the GPU. Every

```
select   *
from     R1,R3,
         (select   R2.z,count(*)
         from      R2
         where     R2.y=3
         group by R2.z) R2
where    R1.x=7 and R1.a=R3.b and R2.z=R3.c
```

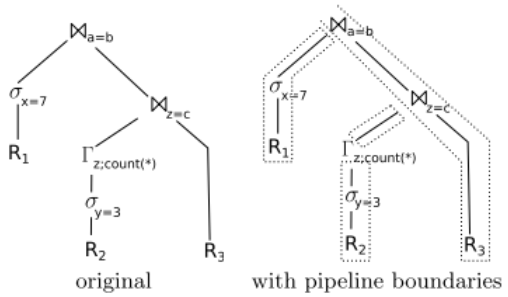



Figure 5.2.: Example for Neumann's merging of operators [72]

operator requires synchronization and result materialization. On the GPU, where the latency to global memory is very high and size is even more limited than system memory, this poses unacceptable problems.

The model we proposed in [88] is similar to the concepts of Neumann [72] and Krikellas et al [60]. Both already mention parallelism as future work, but do not provide details on this topic. In contrast to them, the authors of [19] focused on building code for parallel execution on the CPU. In the next section we show that we can use a set of patterns based on their work to build code that needs only one global synchronisation point—perfectly suited for parallel execution on the CPU—and avoids materialization of intermediate results. By adding a second parallelisation layer that works in a SIMD-fashion, we extend this model to the GPU. In the end we can even conclude that the extended model works well on both platforms even with the same code-basis.

## 5.3. A Model for Parallel Query Execution

In this section we describe a model for parallel execution of relational operators on the CPU. This model targets a MIMD-architecture, where threads can execute different instructions. One of the main challenges when designing parallel algorithms is to keep the amount of communication between threads low, because communication requires threads to by synchronized. Every time we synchronize threads to exchange data, some cores are forced to wait for others to finish their work. One strategy to avoid synchronization is to partition the work in equally-sized chunks and let every thread process one or more chunks independently. In the end, the results of each thread are merged into one final result. The less time is spent on partitioning, the more can be used to do the real work. This approach is called bulk-synchronous model [101] and describes the first level of parallelism we need to distribute work over independent threads.

Dees et al. [19] have shown that we can create code for most queries that fits into such a concept. The query execution always starts with a certain table (defined by the join order), which is split horizontally. The split is performed only logically and does not imply real work. In the first phase of the query execution every partition of the table is processed by one thread in a tight loop, i.e., selections, joins, and arithmetic terms are executed for each tuple independently before the next tuple is processed. We call this

the *Compute* phase, since it is the major part of the work. Before we start the second phase of the query execution, called *Accumulate*, all threads are synchronized once. The Accumulate phase merges the intermediate results into one final result.

This model is similar to the MapReduce model, which is used to distribute work over a heterogeneous cluster of machines [18]. However, our model does not have the limitations of using key-value pairs. Additionally, the Accumulate phase does not necessarily reduce the number of results, while in the MapReduce model the Reduce phase always groups values by their keys. In case of a selection, for instance, the Accumulate phase just concatenates the intermediate results.

The Accumulate phase depends on the type of query, which can be one of the following patterns.

#### Single Result

The query produces just a single result line, e.g., a SQL query with a `sum` clause but no `group by`, TPC-H query 6 for instance:

```
select sum(l_ext...*l_discout) from ...
```

This is the simplest case for parallelization: Each thread just holds one result record. Two interim results can be easily combined to one new result by combining the records according to the aggregation function.

#### (Shared) Index Hashing

On SQL queries with a `group by` clause where we can compute and guarantee a maximum cardinality that is not too large, we can use index hashing. This occurs for example in TPC-H query 1:

```
select sum(...) from ... group by l_returnflag, l_linestatus
```

We store the attributes `l_returnflag` and `l_linestatus` in a dictionary-compressed form. The size of each dictionary is equal to the number of distinct values of the attribute. With this we can directly deduce an upper bound for the cardinality of the combination of both attributes. In our concrete example the cardinality of `l_returnflag` is 2 and `l_linestatus` has 3 distinct values for every scale factor of TPC-H. Therefore the combined cardinality is $2 \cdot 3 = 6$. When this upper bound multiplied with the data size of one record (which consists of several sums in query 1) is smaller than the cache size, we can directly allocate a hash table that can hold the complete result. We use index hashing for the hash table, i.e., we compute one distinct index for the combination of all attributes that serves as a hash key. On the CPU, we differentiate whether the results fit into the L3-(shared by all cores of one CPU) or the L2-(single core) cache. If only the L3 cache is sufficient, we use a shared hash table for each CPU where entries are synchronized with latches. We can avoid this synchronization when the results are small

enough to fit into L2 cache. In this case we use a separate hash table for each thread while remaining cache efficient.

For combining two hash tables, we just combine all single entries at the same positions in both hash tables, which is performed similar to the single result's handling. There is no difference in combining shared or non-shared hash tables.

#### Dynamic Hash Table

Sometimes we cannot deduce a reasonable upper bound for the maximum `group by` cardinality or we can not compute a small hash index efficiently from the attributes. This can happen if we do not have any estimation of the cardinality of one of the `group by` attributes or the combination of all `group by` attributes just exceeds a reasonable number. In this case we need a dynamic hash table that can grow over time and use a general 64-bit hash key from all attributes. The following is an example query in which it is difficult to give a good estimation for the `group by` cardinality:

```
select sum() ... from ... group by l_extendedprice,
n_nationkey, p_partsupp, c_custkey, s_suppkey
```

Combining two hash tables is performed by rehashing all elements from one table and inserting it into the other table, i.e., combing the corresponding records.

#### Array

For simple `select` statements without a `group by`, we just need to fill an array with all matching lines. The following SQL query is an example for this problem:

```
select p_name from parts where p_name like '%green%'
```

## 5.4. Extending the Model for GPU Execution

In this section we explain what is necessary to use the Compute/Accumulate model for query execution on GPUs. Since only one global synchronization point is needed, we can execute each phase in one kernel. As we can see in Figure 5.3(a), the workgroups of the *Compute* kernel work independently and write the intermediate results (1) to global memory. This is similar to the threads executed on the CPU. The *Accumulate* kernel works only with one workgroup. It writes the final result (3) to global memory (the GPU's RAM).

Until now, nothing has changed in the model. To differentiate it from the extended model that we introduce in a moment, we call this the *global Compute/Accumulate* process. It is a direct translation of the CPU implementation and shows that workgroups on the GPU are comparable to threads on the CPU.

However, in contrast to CPU threads, every OpenCL workgroup is capable of using up to 1024 *Stream Processors* for calculations. Threads running on those Stream Processors do not run independently but in a SIMD fashion, i.e., at least 32 threads are executing

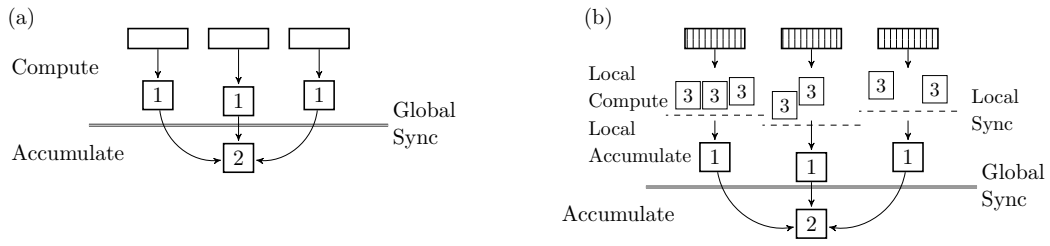



Figure 5.3.: (a) Basic Model, (b) Extended Model
Legend: 1 – intermediate results, 2 – global result, 3 – private result

the same instruction at the same time. These threads form a so called *warp*—the warp size may be different for future architectures. If there are branches in the code, all threads of one warp step through all branches. Threads that are not used in a certain branch execute No Operations (NOPs) as long as the other threads are calculating. The threads of one warp are always in sync, but all threads of one workgroup must be synchronized manually if necessary. Although this local synchronization is cheap compared to a global synchronization, we have to be careful about using it. Every time it is called, typically all workgroups execute it. Since we use up to 1,000 workgroups (see Section 5.5.4), the costs for one synchronization also multiply.

Therefore we apply a *local Compute/Accumulate* model in the *global* Compute phase. First, the local execution is adapted to the GPU's memory hierarchy. Because the local results are only accessed by the threads of one workgroup, there is no need to write them to global memory. The whole process can be seen in Figure 5.3(b). Every (GPU-)thread computes its result in private memory (3), usually the registers of each Stream Processor. At the end of the local Compute phase, the results are written to local memory, the threads are synchronized, and the local Accumulate phase is started.

Second, in contrast to the independent workgroups of the global model, the local processes work in a SIMD fashion and therefore differ from the global phase in the detail. Since we do mostly the same with every tuple we can do this in parallel on a batch of tuples at a time in a SIMD-fashion. The size of the batch equals the number of threads per workgroup. Instead of splitting the partitions in contiguous chunks like in the global model, we stripe them. The thread ID equals the position of the tuple in the batch, i.e., the first thread of the workgroup works on the first tuple, the second on the second, and so on. This works much better than horizontally partitioning the table, because we enforce coalesced memory access. The worst case for this SIMD approach arises if a few tuples of the batch require a lot of work and the rest is filtered out early. As we explained above this leads to a lot of cores executing NOPs, while only a few are doing real work. If all tuples of the batch are filtered—or at least all tuples processed by one warp—the GPU can skip the rest of the execution for this batch and proceed to the next. Fortunately this is often the case in typical OLAP scenarios data, e.g., data of a certain time frame is queried and is stored in the same part of the table.

The aggregation specified by the query is in parts already done in the local Compute phase, where each thread works on private memory. In the local/global Accumulate

Table 5.1.: SIMD operations required by pattern

| Query result pattern | SIMD Operation | Sort |
|---|---|---|
| Single Result | sum | bitonic |
| Index Hashing (group by) | sum | bitonic |
| Array | prefix sum & copy | merge |

phase we merge these private/intermediate results. The threads are writing simultaneously to local memory, so we require special algorithms that work in a SIMD fashion and avoids memory conflicts. The algorithms (prefix sum, sum, merge, and bitonic sort) are well known and can be found in the example directory that is delivered with NVidia's OpenCL implementation. Therefore we just name them at this point and give a short overview but not a detailed description.

The easiest case is no aggregation at all, where we just copy private results to local memory and later to global memory. However, since all threads copy their result in parallel, the right address has to be determined to guarantee that the global result has no gaps. Therefore, each thread counts the number of its private results and writes the size to local memory. To find the right offset for each threads starting position we calculate the prefix sum in local memory. This is done by pairwise adding elements in $\log n$ steps for $n$ being the number of threads. In the end every position holds the number of private results of all threads with smaller IDs and therefore the position in the global result.

We use a very similar approach for the `sum`-Aggregation. In case of a single result pattern (see Section 5.3) we calculate the sum of the private results in parallel by again using pairwise adding of elements in local memory. For index hashing (see Section 5.3) this has to be done for each row of the result. The second column of Table 5.1 shows the SIMD operation belonging to the results pattern.

Sorting is the final process of the Accumulate phase. In every case we can use bitonic sort. In case of the array pattern it is also possible to pre-sort the intermediate results in the Compute phase and use a simple pair-wise merge in the Accumulate phase.

### 5.4.1. Concrete Example

Figure 5.1 shows the OpenCL code for the Compute kernel of query 4 (Q4):

```
select o_orderpriority, count(*) from orders
where o_orderdate >= '1993-07-01'
and o_orderdate < '1993-10-01'
and exists (select * from lineitem
 where l_orderkey = o_orderkey
 and l_commitdate < l_receiptdate)
group by o_orderpriority
order by o_orderpriority
```

Parameters for the kernel are pointers to the columns in global memory, a pointer to the result memory and the size of the block that is processed by each thread. The query parameters, such as the date and the number of expected results, are compiled into the code at run-time. We use `WG_SIZE` as abbreviation for `get_local_size(0)`, which is the number of threads per workgroup. Before starting the local Compute phase private (per thread) and local (per workgroup) memory segments are initialized and every thread calculates its starting position.

The actual Compute phase is represented as a for-loop. Inside the loop the tuples are filtered according to the `where` clause, which is a date range in Q4. The inner loop after the filter instructions represents the join operation between *lineitem* and *orders*. In this loop we iterate over the join index and filter tuples according to the inner `select`'s `where` clause. As soon as one tuple is found, the private counter for this *orderpriority* is incremented and the rest of the inner loop is skipped. At the end of the Compute phase every thread of a workgroup holds the result for the tuples it processed in private memory.

In the local Accumulate phase these results are added to the interim result of every workgroup. A `sum` function as described in Section 5.4 is executed on every line of the result to sum up the counted tuples of the Compute phase.

Listing 5.1: Compute kernel of Q4

```
1  __kernel void global_compute (
2      __global unsigned short* p_c1,
3      ... //columns/indexes as parameters
4      __global long* interim_result,
5      int bs//num of tuples processed by 1 thread
6      ) {
7    __local long l_thread_data[RESULT_SIZE];
8    long p_thread_data[RESULT_SIZE];
9    ... //init memory with 0
10   uint start = (get_group_id(0) * WG_SIZE * bs)
11                + get_local_id(0);
12   uint end = start + WG_SIZE * bs;
13 /* local compute */
14   for (unsigned i0=start; i0<end; i0+=WG_SIZE)) {
15     unsigned short c1 = p_c1[i0];
16     if (!(c1 >= 8582)) continue;
17     if (!(c1 < 8674))  continue;
18     unsigned c0 = p_c0[i0];
19     unsigned ind1e = ind1[i0 + 1];
20     for (unsigned i1=ind1[i0]; i1<ind1e; ++i1) {
21       if (!(p_c2[i1] < p_c3[i1])) continue;
22       ++p_thread_data[c0];
23       break;
24     }
25   }
26 /* local accumulate */
```

```
27    for (int j=0; j < RESULT_SIZE; ++j) {
28      local_sum(&l_thread_data[j],p_thread_data[j]);
29    }
30    ... //copy l_thread_data to interim_result
31  }
```

### 5.4.2. Limitations of the GPU Execution

In this section we discuss the limits of our framework. We categorize these in two classes:

**Hard limits** Due to the nature of our model and the GPU's architecture there are limits that cannot be exceeded with our approach. One of these limits is given by OpenCL: we cannot allocate memory inside kernels, so we have to allocate memory for (intermediate) results in advance. Therefore we have to know the maximum result size and since it is possible that one thread's private result is as big as the final result, we have to allocate this maximum size for every thread. One solution to this problem would be to allocate the maximum result size only once and let every thread write to the next free slot. This method would require a lot of synchronization and/or atomic operations, which is exactly what we want to avoid. With our approach we are limited to a small result size that is known or can be computed in advance, e.g., top-k queries or queries that use `group by` on keys that have a relatively small cardinality. The concrete number of results we can compute depends on the temporary memory that is needed for the computation, the hardware, and the data types used in the result, but it is in the order of several hundred rows.

**Soft limits** Some use cases are known not to fit to the GPU's architecture. It is possible to handle scenarios of this type on the GPU, but it is unlikely that we can benefit from doing that. The soft limits of our implementation are very common for general-purpose calculations on the GPU. First, the GPU is very bad at handling strings or other variable length data types. They simply do not fit into the processing model and usually require a lot of memory. An indication for this problem is, that even after a decade of research on general-purpose computing on graphics processing units and commercial interest in it, there is no string library available for the GPU, not even simple methods such as `strstr()` for string comparison. In most cases we can avoid handling them on the GPU by using dictionary encoding. Second, the size of the input data must exceed a certain minimum to use the full parallelism of the GPU (see Section 5.5.4).

Furthermore, GPUs cannot synchronize simultaneously running workgroups. Therefore we cannot efficiently use algorithms accessing shared data structures, such as the shared index hashing and the dynamic hash table approach described in Section 5.4.

## 5.5. Evaluation

In this section we present the performance results of our framework, which have in parts already been shown in [88]. Each query consists of two functions that are compiled at run-time when the query is executed. On the CPU TBB is used to execute the functions in parallel, the function code itself is compile with LLVM. On the GPU every query requires two kernels, OpenCL is used for parallelization as well as for JIT compilation.

### 5.5.1. Details on Data Structures

Our data structures are kept very simple and optimized for columnar main memory access. The basic data structure is an array for each column where all values are stored consecutively. We use only basic data compression, the number of bytes used for numeric values is the smallest integral C++ type that can hold all the values of the array, i.e., either $1, 2, 4$ or $8$ bytes. For variable-length strings we only store a pointer directing to the underlying character array. However, we seldom store strings directly in a single array, but use dictionary encoding for string columns instead, because handling strings—especially with SIMD instructions and on the GPU—cannot be done efficiently. These dictionaries hold all distinct values in sorted order. This is advantageous as ordering properties remain valid for the integers encoding the values in the dictionary. With this, sorting or range lookups can be directly performed on the integers. Having a dictionary for a certain column, we also know the number of distinct values. In case of *group by*-queries, we can often use this knowledge to allocate the correct (maximum) amount of memory for the result. We also keep a minimum of statistical information: For numeric columns we keep track of the minimum and maximum value residing in that column. This is useful to perform efficient and correct casting for calculations, e.g., in a `sum` clause.

Another important part of our data structures are join indices. We generate join indices for all foreign key declarations in both directions. Instead of traditional B-tree indices we use position-based join indices. Therefore we implemented several variants depending on join cardinality. For example, columns referencing a column of another table via a foreign key are guaranteed to have exactly one matching row. Here our join index consists of a simple array holding single row numbers referencing the target table. For columns where each row references several rows of the joined table, we store a list of referenced row numbers. The list is stored by holding a begin and end offset pointing to a data array with the row numbers. While these join indices may cause a lot of random accesses for certain relations even if we sort the numbers within each list, the simple access patterns usually pay off. In effect, these simple access patterns make it possible to use indices on the GPU in the first place.

### 5.5.2. Test System and Test Data

To evaluate our implementation we run several different TPC-H queries on the scale factor 10 dataset (10 GB data). We compare the performance of NVidia's Tesla C2050

to a HP Z600 workstation with two Intel Xeon X5650 CPUs. Details about the hardware, such as GFLOPs, are listed in Appendix A.1. However, floating point operations are rarely needed in query execution and to the best of our knowledge there are no comparisons on the performance of integer operations for our devices. All tests were conducted on Ubuntu 12.04 LTS. The framework was compiled with gcc 4.5 and TBB 3.0 Update 8. The JIT compilation was done by LLVM 2.9 for the CPU and the OpenCL implementation that is shipped with CUDA 5.0 for the GPU.

We performed our experiments on seven different TPC-H queries. We ran all of those queries on the CPU as well as on the GPU, i.e., all of those queries are suitable for GPU execution. Therefore, we can either use the Single Result (Q6) or the index hashing (all other queries) pattern as described in Section 5.4. By selecting queries with different characteristics we tried maximizing the coverage for different query types. For example, the queries vary in the size of the result structure, in the number of tables involved or in the kind of operator patterns.

### 5.5.3. GPU and CPU Performance

Since the native CPU implementation described in Section 5.3 was already compared to other DBMS [19], we take it as our baseline and compare it to our OpenCL implementation.

In our first experiment we compare the implementations by measuring the raw execution time of each query. As explained in Section 5.1 the necessary columns for execution are in memory, we do not consider the compile times, and we include result memory allocation (CPU and GPU) and result transfer (only needed on the GPU). We configure the OpenCL framework to use 300 workgroups with 512 threads each on the GPU and 1000 workgroups with 512 threads on the CPU. This configuration gives good results for all queries as we show in Section 5.5.4.

Figure 5.4 compares the execution times for seven TPC-H queries on different platforms: Z600 is the native CPU implementation on the Z600 machine, Tesla/CL the OpenCL implementation on the Tesla GPU and Z600/CL the OpenCL implementation executed on the CPU of the workstation. As we can see, the OpenCL implementation for the GPU is considerably faster than the native implementation on the workstation for most queries. The only exception is Q1, for which the GPU takes almost twice as long. The OpenCL implementation of Q1 on the CPU is significantly worse than the native implementation as well. The reason for this is hard to find, since we cannot measure time within the OpenCL kernel. The performance results might be worse because in contrast to the other queries the execution of Q1 is dominated by aggregating the results in the local accumulate phase. This indicates that the CPU works better for queries without joins but with aggregations. Further tests and micro-benchmarks are needed to proof this hypothesis.

It is remarkable that the OpenCL implementation on the CPU achieves almost the same performance as the native CPU implementation for half of the queries, considering that we spent no effort in optimizing it for the actual architecture.

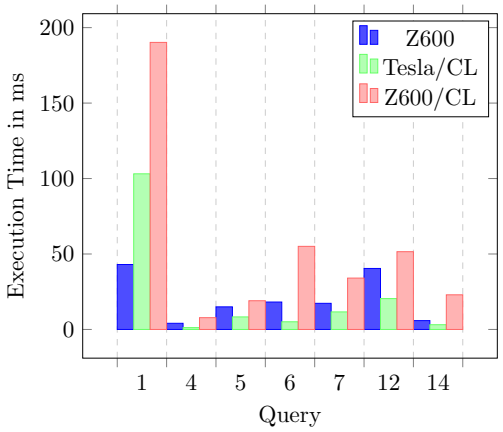



Figure 5.4.: Absolute execution times

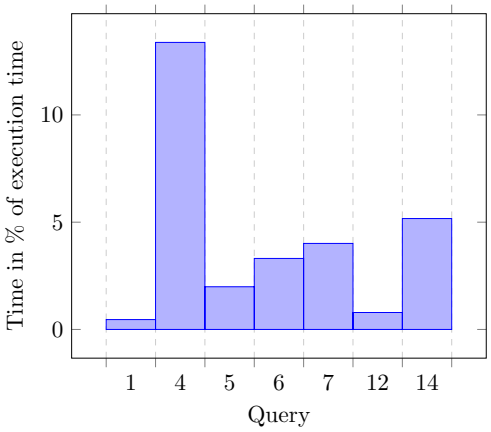



Figure 5.5.: Transfer time for final results in comparison to execution time

### 5.5.4. Number of Workgroups and Threads

We illustrate in Figure 5.6 how the number of workgroups used for the execution of the query influences the performance. We measure two different configurations: the left figure shows the results when we use 512 threads per workgroup; on the right we use 256 threads per workgroup. Due to implementation details our framework is not able to execute queries 1 and 5 for less than around 140 resp. 200 workgroups. For the other queries we can clearly see in both figures that we need at least around 50,000 threads in total to achieve the best performance. This high number gives the task scheduler on the GPU the ability to execute instructions on all cores (448 on the Tesla C2050) while some threads are waiting for memory access. There is no noticeable overhead if more threads are started with the following exceptions:

Query 4 shows irregular behavior. It is the fastest query in our set, the total execution time is between 1 and 2 ms. Therefore uneven data distribution has a high impact.

Query 5 is getting slower with more threads, because the table chunks are too small and therefore the work done by a thread is not enough. The table, which is distributed over the workgroups, has only 1.5 mio rows, i.e., each thread processes only 3 rows in case of 1,000 workgroups with 512 threads each. This matter is even worse with Query 7, because the table we split has only 100,000 rows. The other queries process tables which have at least 15 mio rows.

This experiment shows that the GPU works well with a very high number of threads as long as there is enough work to distribute. In our experiments, one thread should process at least a dozen rows.

The behavior is similar to task scheduling on the CPU, where task creation does not induce significant overhead—in contrast to thread creation. Figure 5.7 shows the execution time of the native implementation depending on the *grain sizes*, which is the TBB term for the size of the partition and therefore controls the number of tasks. As we

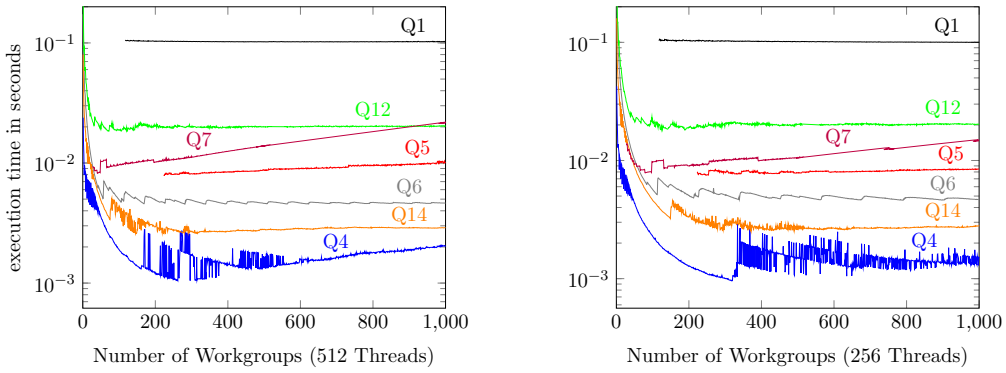



Figure 5.6.: Comparison of different workgroup numbers (left: 512, right: 256 threads per workgroup)

can see, a large grain size prevents the task scheduler from using more than one thread and therefore slows down execution. The optimal grain size for all queries is around 1000. We achieve a speed-up of 7 up to 15 for this grain size. If the partitions are smaller, the execution time increases again. Similar to our OpenCL implementation the optimal number of tasks per core is around 100.

### 5.5.5. The Overhead for Using the GPU

Although the data to be processed is already accessible by the GPU, we have some overhead for calling the kernels, allocating the memory, and transferring the result back to main memory in the end. Depending on the amount of memory that needs to be allocated and transferred, this takes between 0.2 and 1.0 ms. In Figure 5.5 we show the impact on the execution time. Especially for the short-running queries 4, 6 and 14 the time needed is noticeable. Although we need only two kernel calls and transfer not more than a few kilobytes for the result, the overhead makes almost 10 percent.

This shows that it is very important to communicate between CPU and GPU as rarely as possible. If we called each operator in an Open-Next-Close-fashion or had to reserve memory for materialized results of each operator, the overhead would take more time than the actual query execution.

## 5.6. Related Work

Over the last decade there has been a significant amount of research on using GPUs for query execution. He et al. built GDB, which is able to offload the execution of single operators to the GPU [42]. They implemented a set of primitives to execute joins efficiently. The main problem is that the data has to be transferred to the GPU, but only a small amount of work is done before copying the results back to the main memory. Peter Bakkum and Kevin Skadron ported SQLite to the GPU [6]. The complete database is stored on the graphics card and only the results are transferred. Since all operations

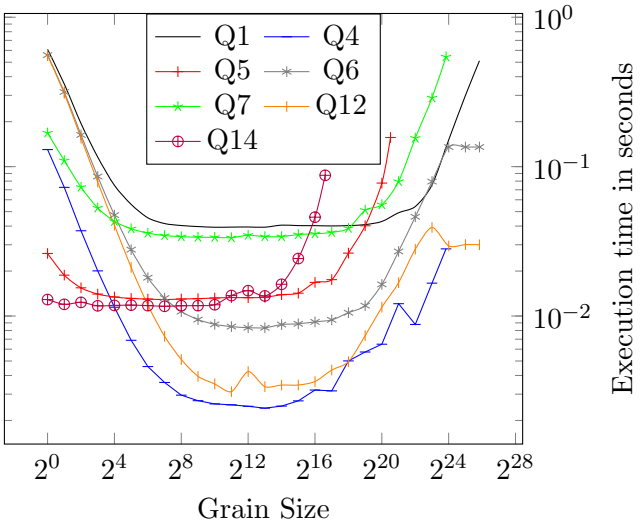



Figure 5.7.: Comparison of different grain sizes

are executed on the GPU, their system is not able to use variable length strings. It can process simple filter operations, but does neither support joins nor aggregations. Another approach with the focus on transactional processing has been tried in [44]. Their implementation is able to execute a set of stored procedures on the data in the GPU's memory and handles read and write access.

A very good summary about published database operators such as selection, joins and aggregations on external GPUs is given by the authors of [46]. Until now only one publication about an operator on an integrated CPU/GPU architecture is available: in [54] the authors propose a stream join for the GPU which makes use of the shared memory provided by AMD's Trinity architecture. Since it is not necessary to transfer data anymore, the HELLS-join is depending only on the processing power and on the bandwidth between the processing unit and the memory. The evaluation shows that Trinity's GPU allows faster stream join processing than it's CPU.

The most promising approach to execute analytical queries on heterogeneous hardware has been proposed by Heimel et al [46]. They re-implemented all relational operators available in MonetDB with OpenCL and showed that this delivers the same performance on the CPU as MonetDB's native multi-core-CPU approach. The benefit is of course that OpenCL can be executed on other supported processor architectures. The approach targets heterogeneous processors, such as AMD's APUs, but can also execute operators on an external co-processor. To hide the transfer they cache columns on the GPU. Their results show—similar to ours—that a modern GPU can be faster than the CPU in many cases if you ignore the time needed to copy the data to the external memory. However, there are also queries that run faster on the CPU.

The authors of [106] create OpenCL and CUDA code for OLAP queries of the Star Schema Benchmark (SSB) [78]. Their paper is written under the assumption that

the GPU is better at query execution than the CPU (Yin), but transfer is necessary (Yan). Consequently they integrated the PCIe transfer into their model and showed some promising ways to use streaming for star schema queries. Their evaluation shows that every query runs faster on their framework than in MonetDB—even with transfer. From our point of view this is an unfair comparison. While their framework is able to handle only SSB queries, Monet DB is a full-fledged DBMS that can execute any type of query and adheres to the ACID rules. They tuned their implementation manually, e.g., with the help of invisible Joins, but left MonetDB in its original state. As for our work a comparison with the implementation of Dees et al. [19] or Neumann [72] would be more appropriate. Nevertheless, their streaming approach may in many cases be a good alternative for data replication as we used it and becomes even more interesting with heterogeneous platforms.

Compiling a complete query plan (or parts of it) into executable code is a radical and novel approach. Classical database systems generate executable code fragments for single operators, which can be used and combined to perform a SQL query. There are several recent approaches for generating machine code for a complete SQL query. The prototype system HIQUE [60] generates C code that is just-in-time-compiled with gcc, linked as shared library, and then executed. The compile times are noticeable (around seconds) and the measured execution time compared to traditional execution varies: TPC-H queries 1 and 3 are faster, query 10 is slower. Note that the data layout is not optimized for the generated C code. HyPer [56, 72] also generates code for incoming SQL queries. However, instead of using C, they directly generate LLVM IR code (similar to Java byte code), which is further translated to executable code by an LLVM compiler [99]. Using this light-weight compiler they achieve low compile times in the range of several milliseconds. Execution times for the measured TPC-H queries are also fast, still, they use only a single thread for execution.

One of the fundamental models for parallel query execution is the exchange operator proposed by Graefe [35]. This operator model allows both intra-operator parallelism on partitioned datasets as well as horizontal and vertical inter-operator parallelism. Although this model can also be applied to vectorized processing strategies it still relies on iterator-based execution. Thus, the required synchronization between each operator can induce a noticeable overhead.

## 5.7. Conclusion

In this chapter we showed how we can compare the performance of query execution on the GPU with the one on the CPU in a fair manner. In other works we often see two extremes: Either the presented GPU implementation is compared to the performance of a full fledged DBMS that provides much more functionality and adheres to the ACID-rules or the GPU implementation is modified so it can be used on the CPU as well.

In contrast to that we tried to provide a fair comparison by using a hand-tailored solution for the CPU and one for the GPU that both use a comparable model for execution. We are confident that the tested queries cannot be executed faster than

that on the used hardware.

The results show that some queries can indeed be executed faster on the GPU than on the CPU and that OpenCL can be used to parallelize algorithms on the CPU.

# 6. Automatically Choosing the Processing Unit

In the previous chapters we have shown that some tasks in a DBMS can be offloaded to a co-processor with benefit. In many cases however, the CPU is faster than the GPU for certain input sizes or parameters. Therefore, we need a framework that offloads tasks automatically only if this is beneficial. Such a framework—called HyPE—was developed as a joint work of researchers from Otto-von-Guericke University Magdeburg and TU Ilmenau in [12].[1]

In Section 6.1 we motivate, why automatic scheduling of operators is necessary. The concept of operators as we use it, is explained in Section 6.2. In Section 6.3 we present the decision model for offloading operators to a co-processor. HyPE is the implementation of this model. We evaluate it with the help of two use cases in Section 6.4. We take a look at related work in Section 6.5 and conclude this chapter in Section 6.6.

## 6.1. Motivation

In the previous chapters we showed that a DBMS can benefit from a co-processor such as the GPU but not all calculations should be offloaded there. Unfortunately the CPU's and GPU's architectures are much too different, to safely predict if a task runs faster on the CPU or on the co-processor—although there are indicators and rules of thumb. For some tasks, such as the dictionary merge algorithm presented in Section 4.3, there is no benefit in using the GPU at all. In this case the transfer is the dominant factor, but there are also tasks that are not parallelizable in an efficient manner.

In many cases, however, there is a range of input sizes, where it makes sense to execute the task on the GPU (see Sections 3.4.4, 4.2.2, 4.1.2). We call this range *GPU-benefit range* as shown in Figure 6.1. If, on one hand, the amount of input data is to small, the GPU cannot be fully utilized. On the other hand a large input might not fit into the GPU's memory and cannot be processed without swapping. The GPU-benefit range strongly depends on the CPU and GPU in the system; there are three common configurations:

**Desktop configuration** In desktop systems we often find powerful gaming graphic cards combined with a standard desktop CPU, which has two or four cores. In such a system the GPU-benefit range is wide, as long as single precision calculations are sufficient.

[1] The implementation can be found here: http://wwwiti.cs.uni-magdeburg.de/iti_db/research/gpu/hype/

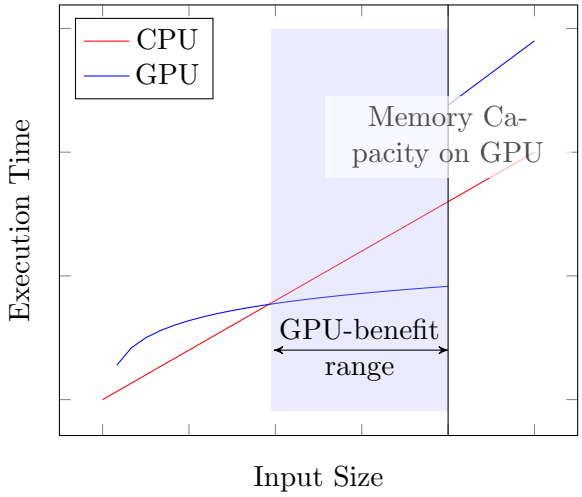



Figure 6.1.: The *GPU-benefit range*

**Server configuration** Server CPUs are typically more powerful than their desktop pendants and have four to eight cores. In Table 6.1 a common combination of processor and graphics card for desktop and server machines is shown.[2] The Xeon processor is able to execute twice the number of operations within the same time than the desktop processor. High-end desktop CPUs have a similar performance like server CPUs. However, workstation or server machines usually have multiple CPUs that are interconnected with a fast network.

Server class GPUs on the other hand do not provide significantly more single precision performance. The K20 for instance is only 10% more powerful than the GTX 770. Instead, vendors like NVIDIA focus on adding performance for double precision calculations: the server GPU processes nine times as many floating point operations as the desktop GPU.

In combination this means that on server machines there are less tasks that can be executed faster with the GPU than with the CPU: the GPU-benefit range is tighter than on a desktop machine.

**Heterogeneous architecture** At the time of writing there are CPUs available that have an integrated GPU, namely Intel's Haswell architecture and AMD's APUs, but they are mainly sold in mobile and energy-efficient desktop PCs. There is a high probability that we will see these heterogeneous architectures in future server CPUs as well. These heterogeneous processors share the main memory—and even some parts of the cache hierarchy—between all processing units [14]. Hence, the transfer bottleneck vanishes.

[2]The listed calculation power in Floating Point Operations per Second (FLOPS) is a theoretical value and should only be used as an indicator. Vendors use different ways of determining the calculation power. It makes no sense to compare the CPU numbers with the GPU numbers, but comparing the devices of one vendor is feasible.

| *Calculation Power in GFLOPS* | CPU | GPU (SP/DP) |
|---|---|---|
| Desktop: i5-3450, GeForce GTX 770 | 99.2 | 3213/134 |
| Server: Xeon E5-4650, K20 | 172.8 | 3520/1173 |

Table 6.1.: Comparison of typical desktop and server configurations. Numbers from hardware specifications [49, 76].

However, this huge advantage brings two disadvantages: First, because they share the same memory, they both use the same memory technology. External GPUs are equipped with GDDR-memory that has a wider bandwidth (around 200 GB/s [76]) and a higher latency. The latency can be hidden by using a high number of threads (see Section 5.5.4). Integrated GPUs do use the standard system memory instead, where the bandwidth is significantly lower. Second, integrated GPUs share the same die with the CPU. Hence, they are limited in space and have to use one cooling system together with the CPU, so they are much less powerful than the ones on external cards.

The GPU-benefit range show in Figure 6.1 looks different for heterogeneous architectures because there is no upper limit depending on the GPU's memory capacity. Still, there must be enough data to distribute over all GPU cores, i.e., for small input sizes the CPU is faster.

Many systems belong to one of these three categories, but there are of course also mixtures such as desktop systems with high end CPUs or with both, an integrated and an external GPU. Because of this variety it is not feasible to decide in advance which processing unit to use for a task with a certain input size. Calibrating a machine once is not practical as well because there might be thousands of tasks and for some the GPU-benefit range also depends on parameters other than the input size, e.g., for K-Means the range strongly depends on $k$ (see Section 4.1.2). Hence, a self-tuning approach that decides automatically where to execute certain tasks based on prior knowledge is necessary.

## 6.2. Operator Model

### 6.2.1. Base Model

For the framework we concentrate on static tasks—like the ones shown in Chapter 4—that can either be executed by the CPU or a co-processor such as a GPU. These tasks, which we refer to as operators in the following, are the base granularity for a scheduling decision. Using the framework for query plans generated just in time is limited. Predicting the execution time of a query that is executed in two phases as described in Chapter 5 is nearly impossible, because the influence of certain parts of the query is unknown when they are not modeled in independent operators. However, the scheduling framework can be used if the DBMS has to answer only a limited set of queries—similar to the parametrized queries of the TPC-H set. In this case we treat every possible query

with its two kernels as one operator.

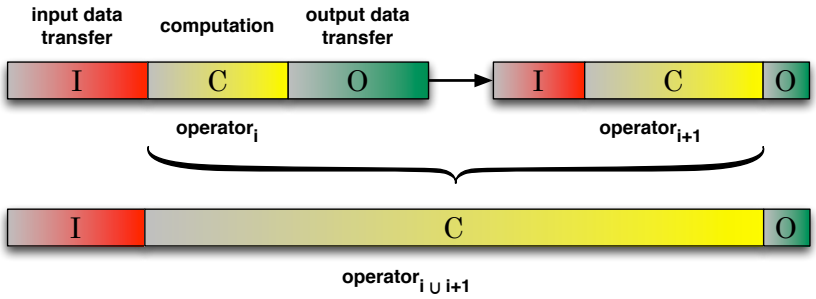



Figure 6.2.: Operator model [12]

An operator consists of three phases: the input data access/transfer, the actual computation time, and the result/output transfer. The output of one operator can be the input for another one as depicted in Figure 6.2. For accessing the input data and storing results, a certain amount of time is required, e.g., I/O time for fetching data from disk, or transferring data from/to an attached co-processor device. These times can easily be measured in current systems or calculated, e.g., when the bus bandwidth and transfer rates of the underlying hardware are known. For actually solving the task, computation time is consumed by the involved processor. This time depends on

1. the algorithm itself
2. the hardware used
3. the size of the input data
4. the characteristics of the input data to be processed, e.g., the selectivity of a complex filter
5. the parameters for the operator, e.g., a value range for a filter condition or more complex parameters, such as the $k$ for the K-Means algorithm

These dependencies are usually first known at run-time and may change. Hence, the computation time is harder to estimate than transfer times. Further, the computation time includes hidden data access times like main memory or cache accesses. Those internal times are hard to measure when the algorithm is executed and usually require sophisticated profiling tools that utilize dedicated system-specific hardware counters.

We regard operators as primitives where the computation phase is an atomic part that starts right after the input data is available. This perfectly matches the kernel model used by OpenCL and CUDA as described in Section 3.3 where input data needs to be transferred to the device before a parallel kernel starts processing, and its output has to be transferred back to the host if it cannot be reused by another kernel. This model is common in literature, e.g., [42]. Of course there are also more complex access patterns, which overlap data accesses with computation, e.g., the 4-way-concurrency

pattern described in Section 3.4.3. Further, several algorithms cannot be expressed with a single atomic computation phase. For instance, within a hash join operator, the probe phase needs to wait until all keys have been inserted into a hashtable. In such cases, the operator could be split further into dependent primitives. In general, this could be done and a scheduling decision could be made for each resulting primitive, but in most cases this approach is counterproductive. First, it complicates the decision because many possible execution plans are generated. Second, splitting such operators that are tightly coupled will hardly result in any improvement with separate processing decisions. Most likely, data transfer costs will negate any performance improvements or even lead to slowdowns. For our model we merge the primitives belonging to one operator as depicted in Figure 6.2. The framework makes a single scheduling decision and all internal data transfers are hidden in the computation phase.

### 6.2.2. Restrictions

There are restrictions to the scheduling decision: for the execution time estimation we neither consider system load nor do we analyze input data to gain knowledge about its characteristics. We explain the reasons and impacts of our framework design in the following.

#### System Load

Although the load of the processing unit may be important for the scheduling decision, it unfeasible to take it into account because of:

- *Model Complexity:* There are numerous resources that are affected by a running task, e.g., memory bandwidth and capacity, caches, processing capacity.[3] Modeling the influence of different tasks on each other is, however, very complex. Moving a data-intensive task to the GPU for instance seems to be a good idea, when the CPU is working on another data-intensive task. Unfortunately the data transfer to the GPU also requires bandwidth from the memory bus and would also slow down the CPU execution. If both tasks accessed the same data, they may even benefit from running together on the CPU because the cache hierarchy can be utilized. Also, just marking a task as data-intensive or compute-intensive may be an over-simplification.

- *Measuring Complexity:* For some parts of the system it is not possible to measure the free resources easily. Determining the available memory bandwidth can only be done by trying or using a central instance to give away resources. A central instance of course only controls its own tasks. Resources used by other processes are left out. Even if the resource usage can be measured, it is only a picture for the moment. The system might look totally different as soon as the scheduling

[3] Tasks requiring network or disk I/O are usually not suitable for a co-processor, because I/O can in general only be handled by the CPU.

decision has been made, for instance because a memory consuming OLAP query just finished.

- *Resource Control:* Controlling resources for one task is very limited. In case of CPU load the only possibility is to limit the number of threads to be used. For the GPU not even this is possible. Another problem is that many tasks use libraries that work as an abstraction layer such as Intel's TBB, which partly controls resource usage itself. Here the number of TBB-tasks does not say anything about the number of threads on the machine being used, because TBB also decides on scheduling tasks on processing units. Hence, the only way to save control the load would be to either execute a task or wait.

#### Data Characteristics

Execution times and result sizes often depend on the input data characteristics such as cardinality, clusters, or (partly) sorted blocks. However, tasks depend on certain characteristics only. Hence, there is no general assumption we can make about execution times for every task. One possibility to include this factor into the scheduling decision is to analyse the data and provide the result as parameter for the operator. The influence of the parameter is then learned by our framework.

## 6.3. Decision Model

In Section 6.1 we discussed that a framework is needed to decide at run-time which processing unit to use. The framework treats the underlying system hardware as black box and uses machine learning techniques to make a decision based on the conditions for the specific execution. Only little setup is required once to adjust the framework to the hardware. As soon as this initial calibration is done it learns the executions times for every task and makes decisions based on the collected data. In this section we describe this framework, which was first presented in [13].

### 6.3.1. Problem Definition

Before the framework decides on the processing unit $p$ the total execution time $T^p_{total}$ of an operator $a$, which solves a problem of class $A$ (e.g., hash join and merge join both solve the join problem), is estimated. $T^p_{total}$ is the sum of in/output transfer time ($T^p_i/T^p_o$) and the actual computation time ($T^p_c$). We assume that there exist (unknown) functions $t^p_i$, $t^p_c$, and $t^p_o$, which depend on different parameters and describe the system behavior for that specific processing unit:

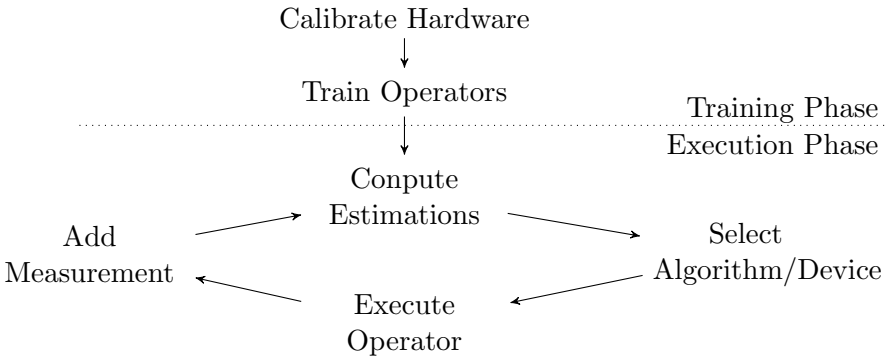



Figure 6.3.: Workload for training and execution phase

$$T^p_{total} = T^p_i + T^p_c + T^p_o \text{ where} \tag{6.1}$$
$$T^p_i = t^p_i(\underline{inputsize}, \underline{HW}, \underline{load}) \tag{6.2}$$
$$T^p_c = t^p_c(\underline{inputsize}, \underline{a}, \underline{HW}, \underline{P}, \underline{load}) \tag{6.3}$$
$$T^p_o = t^p_o(\underline{outputsize}, \underline{A}, \underline{HW}, \underline{P}, \underline{load}) \tag{6.4}$$

As explained in Section 6.2.2 we ignore the dynamic system *load*. The influences of the data size parameters, as well as the system hardware $HW$ are considered to be static assuming that the hardware does not change during run-time. The abstract (multi-dimensional) parameter $P$ describes any number of other dynamic parameters like selectivity or cardinality that influence execution and output transfer time. Because we consider the output of every operator $a \in A$ to be the same, i.e., all algorithms produce the same solution, the output transfer time depends on the class and not on the specific operator. Therefore, the *output size* can be estimated in advance. Only the computation time depends on the specific operator $a$.

Since the actual functions $t^p$ are unknown to the framework, they have to be modeled. This can be achieved with various techniques, e.g., using analytical models, which are rather static, or learning based approaches that require an expensive learning phase (cf. Section 6.5). We use statistical methods that create an initial model with a short training phase and, during run-time, improve it with actual execution times for a specific input parameter set.

### 6.3.2. Training and Execution Phase

The operators' execution models are divided into two phases as shown in Figure 6.3. Before the framework can be used a training phase is needed as initial calibration. First, the static hardware-specific behavior for data transfers needs to be approximated when the system is set up or the hardware configuration changed. Although the theoretical

bandwidth of PCIe 2.0 bus is 8 GB/s, the real bandwidth differs depending on the GPU and the mainboard. Therefore, in a short calibration operation, we copy data blocks with varying sizes to the GPU. By scheduling multiple asynchronous copy transactions we ensure that the available bandwidth is fully utilized. In case asynchronous transfers are not available, e.g., because the memory segments are not pinned (cf. Section 3.4), synchronous transfers are similarly calibrated. For GPU data transfers, we copied data blocks with varying sizes from the host's RAM to the device memory. Since concurrent asynchronous copy operations are supported by modern GPUs, we executed 1000 copy transactions for fully utilizing the available bandwidth. Additionally, synchronous copies, which are serialized by the GPU driver, were scheduled to obtain transfer times that are required for small, single blocks. The bandwidth utilization for a workstation equipped with a Nvidia Tesla C1060 is shown in Figure 6.4.

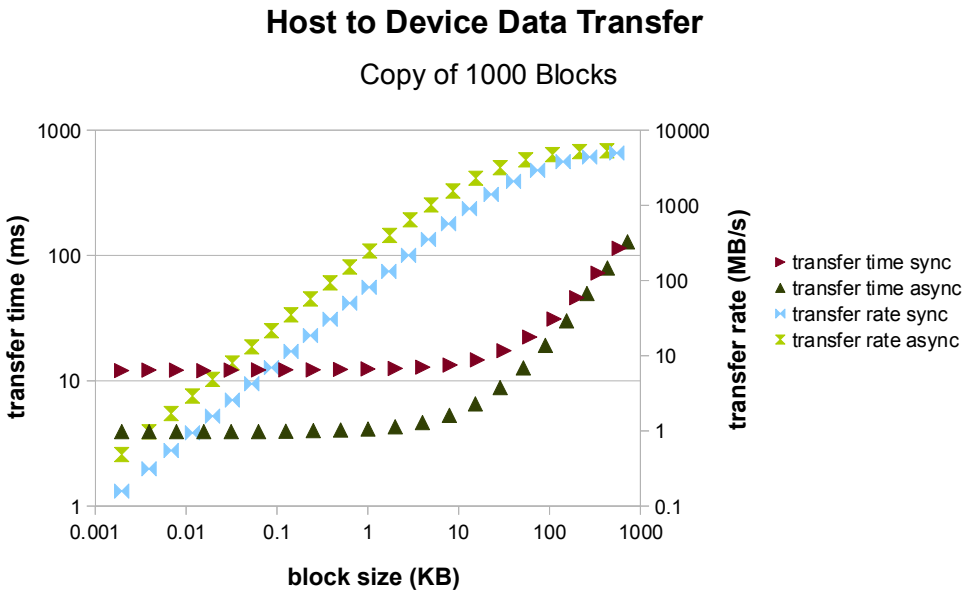



Figure 6.4.: GPU data transfers [12]

The maximal rate we achieved was ≈5.7 GB/s. For block sizes smaller than 10 kB/s the transfer time is dominated by the overhead to execute the copy operation, i.e., 1000 synchronous transfers always take 10 ms no matter the size of each transfer. For synchronous transfers a block size of 0.7 MB is sufficient for fully utilizing the PCIe bus. Because asynchronous transfers use pinned memory and are executed in parallel, which hides the over for calling the GPU driver's copy interface to a certain degree, a smaller block size of around 70 kB is sufficient.

The second step of the training phase is the creation of the initial computation time models for the available algorithms. These models can either be built when the system is set up or at run-time, when operators need to be scheduled. In the latter case no reasonable decision can be made at first; hence the algorithms are just scheduled in a

round robin fashion. The run-time of every operator is measured. Together with the calculated transfer times and the used parameters, these measures form interpolation points that are used to approximate functions describing the computation behavior.

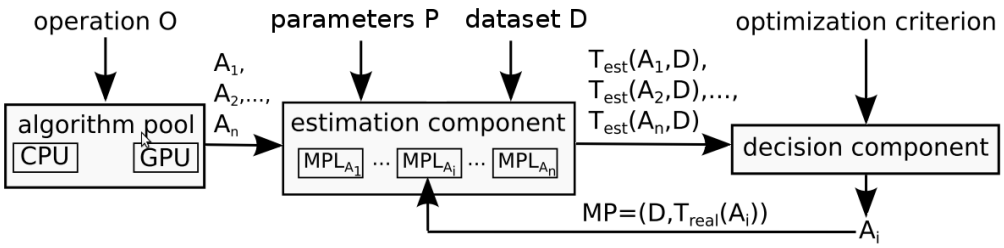



Figure 6.5.: Decision model overview (modified version from [13])

The execution phase starts, when all initial training steps are done. From this point on scheduling decisions are based on the execution times learned in the training phase (Figure 6.5). The processor with minimal expected costs ($T_{est}(a_i, D)$) is chosen and real execution times ($T_{real}(a_i, D)$) are measured. The measurements are then used to update the model in case $T_{est}(a_i, D)$ for the given data set parameters $D$ differs too much from the actual measure $T_{real}(a_i, D)$ similar to [107]. The ($D$, $T_{real}$)-measure pair (MP) is used to update the model accordingly. Details on that can be found in [13].

To limit the amount of memory needed to store measure pairs ring buffers are used to overwrite old measurements. This way the time needed to calculate the approximation functions is also kept low. Using ring buffers causes very low overheads but may lead to worse approximations compared to other aging mechanisms that evaluate the significance for each MP and evict the least important one.

### 6.3.3. Model Deployment

Developing operators used by the decision model is orthogonal to the calibration and execution described in Section 6.3.2. No prior knowledge about the hardware is required to develop operators.

In general, the following steps are required to deploy the framework:

1. **Identify operators**: Similar to the tasks described in Chapter 4 operators have to be identified that are suitable for co-processing. It makes sense to first find hot-spots or bottlenecks during system execution, e.g., with the help of profiling tools, and check if using a co-processor may be promising. In case of the GPU and similar co-processors the points given in Section 3.5.1 can be used as rules of thumb. Then, the task has to be mapped to the operator model introduced in Section 6.2.

2. **Operator implementation**: the algorithm(s) solving the task have to be implemented and tuned for each kind of processing unit that shall be supported. OpenCL can be used for a universal implementation that is supported by the CPU

and potential co-processors. In some cases it may be useful to use frameworks optimized for the given hardware, such as TBB for CPUs or CUDA for NVIDIA GPUs.

3. **Identify scheduling measure**: Until now we described how the scheduling framework can be used to optimize for response time. In some cases it might be useful to make decision base on another criterion such as throughput. The framework can easily be adapted.

4. **Identify parameters**: Next to the input size there might be other parameters that influence the scheduling decision. If these parameters are not given, they need to be estimated, e.g., $k$ in $k$-means is given, the selectivity for a join operator needs to be estimated.

## 6.4. Evaluation

In this section we show that the framework makes accurate decision, i.e., the predicted execution times do not differ too much from the real/measured times. Furthermore, we show that the benefit for using the right processing unit outweighs the overhead for using the framework, i.e., calculating approximation functions and actually making the decision.

### 6.4.1. Use Cases for Co-Processing in DBMS

To evaluate the framework we use two task that we describe in the following: sorting of data and index scans.

#### Data Sorting

We use data sorting as first use case for two reasons: First, the problem of sorting elements in an array has been widely studied and a variety of implementations is available, e.g., with the help of GPUs by Govindaraju et al [33]. Second, sorting is an important primitive for database operations such as sort-merge joins or grouping. It is also part of finding a good compression strategy as described in Section 2.2.3. Furthermore, it is a multi-dimensional problem when using the number of elements to be sorted and the number of CPU cores as parameters during scheduling.

We use the *TBB* sort implementation for the CPU and *Thrust* sort for the GPU. Figure 6.6 shows the speed-up of the GPU over the CPU implementation. As expected depending on the number of available cores on the CPU and the size of the array, sometimes the GPU and sometimes the CPU is faster. We plotted contour lines in the xy-plane that show when a certain speedup value is exceeded for a better orientation. Especially the regions where the speedup is smaller than 1, i.e., the CPU is faster than the GPU are interesting, because our framework should decide for the CPU in this case. In general the CPU is faster when less than 200 000 elements have to be sorted. The more cores are available the higher this value gets. With more than 12 threads, the CPU

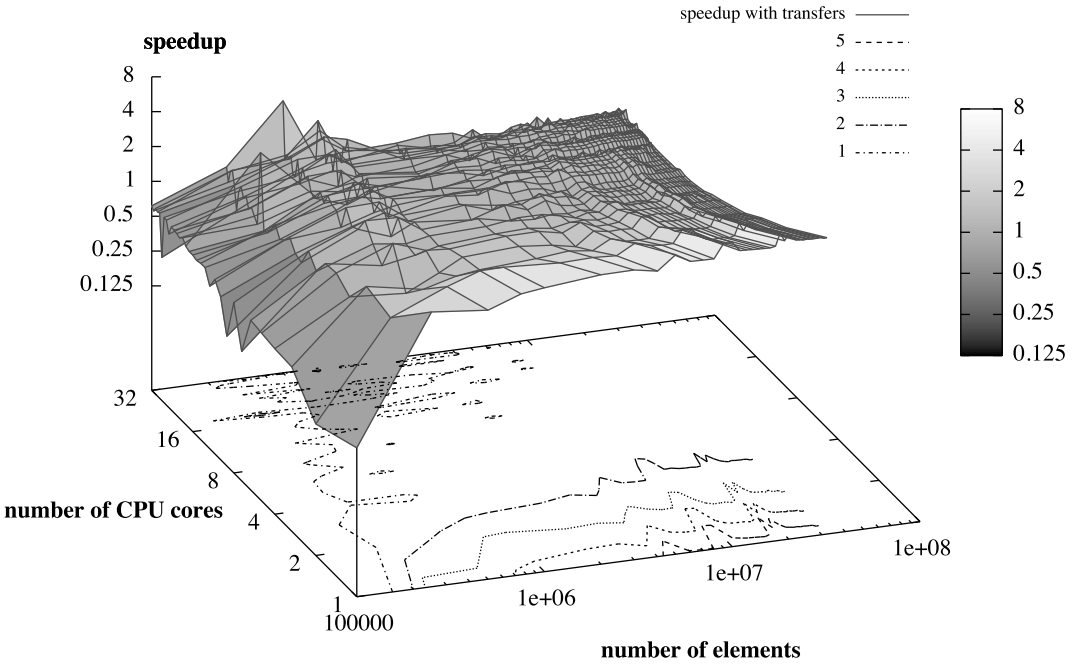



Figure 6.6.: Sorting workload

is still faster at 1 000 000 elements. However, when using many threads the execution time is very unstable, because we depend on the OS to assign resources to our process. In contrast, all cores of the GPU are assigned to the running kernel. Hence, there is almost no variance in the run-time of the kernel for one input size. Also, TBB may use non-optimal partition sizes, which has a high impact on the execution time in case of a small number of elements. We observed similar behavior in the other experiments of this thesis (cf. Section 4.2.3).

### Index Scan

Searching is—next to sorting—another very important primitive often used in DBMS-operations. To speed up search operations, indexes are commonly used on large data sets. Several variants exist for various use cases, e.g., B-trees for searching in one-dimensional datasets where an order is defined, or R-trees to index multi-dimensional data like geometric models. The GiST framework was developed to encapsulate operations on indexes like inserting or removing keys and provides operations for maintaining the tree such as height-balancing [47]. To implement a new index type, only the actual key values and key operations, such as query predicates, have to be defined by the developer. To define an n-dimensional R-tree for instance only minimal bounding rectangles

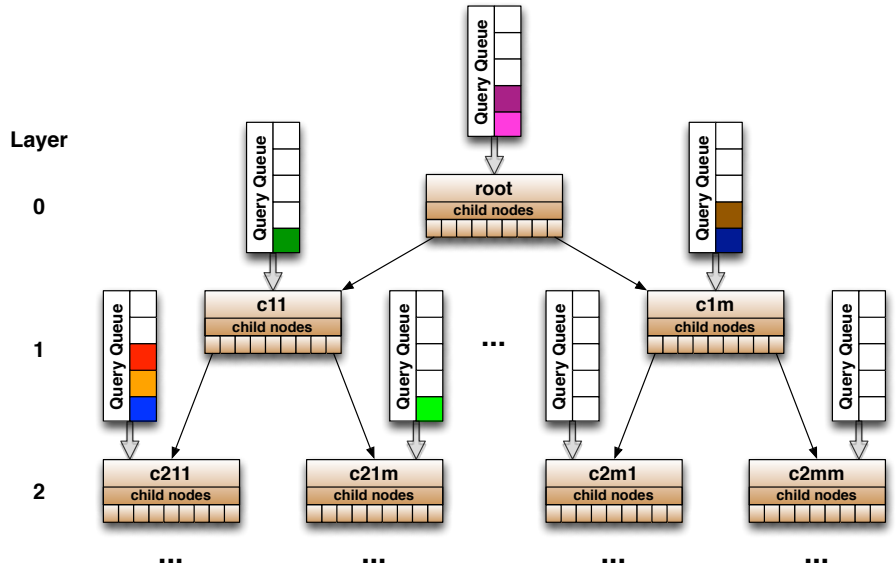



Figure 6.7.: Index tree scan [12]

with n coordinates and an intersection predicate are required. Beier et al. implemented a framework that abstracts the hardware layer and enables look-up operations on the GPU and on the CPU at the same time [7].

The necessary parallelism to benefit from such an approach is achieved by grouping incoming operations and lookup several queries at once. This way two layers of parallelism are created. Every node to be scanned is assigned to one of the CPU's cores or one of the GPU's SMX. On the GPU the query predicates of the batch are then tested against the node's child nodes in parallel on each thread processor.

For optimal scan performance, it is required to determine which node has to be scanned by which (co-)processor. The decision has to be made for every iteration and every node and depends on two parameters. The first one is the node size, i.e., the number of one node's children (*slots*). It is determined once when the index is created and will not change at run-time. Large nodes result in many tests per node but less index levels to be scanned while small nodes reduce the required scan time per node but result in deeper trees. The second parameter is the number of *queries per scan task*, which of course changes with every batch. It depends on the application's workload and the layer where a node resides. The batches start at the root node and are streamed through the tree until they are filtered or reach the leaf layer, returning final results. Hence, the root node and other nodes nearby are tested for almost every query and the number of queries per batch is expected to be large. Near the leaf layer more queries are already filtered out, the parameter is usually smaller here.

The parameters' impact on scan performance is illustrated in Figure 6.8 where the GPU speedup $s = \frac{CPU\ time}{GPU\ time}$ is plotted for different parameter combinations. For small node and batch sizes, the GPU's cores cannot be fully utilized and is up to 2.5 times slower ($= \frac{1}{s}$) than its CPU counterpart. The break even points where CPU and GPU

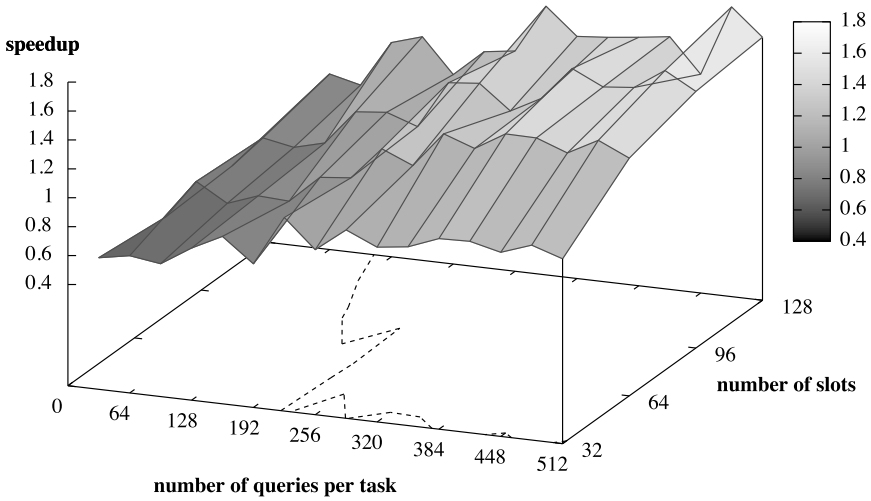



Figure 6.8.: Index scan—GPU speedup [12]

have the same run-time ($s = 1$) are depicted with the dotted line.

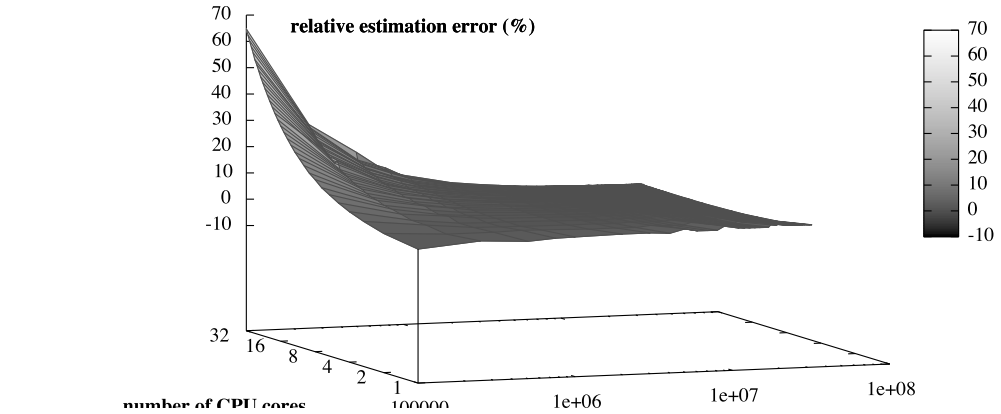
### 6.4.2. Implementation and Test Setup

The decision model described in the previous section was implemented in the HyPE framework. Its estimation component uses statistical methods provided by the *ALGLIB* [4] package. For one-dimensional parameters the *least squares method* and *spline interpolation* provide good estimations while requiring reasonable time for calculations. If more than one parameter influences the execution time of an operator HyPE uses multi-parameter fitting.

We choose the index scan as use case for single-parameter estimations: the number of queries per batch. The number of slots per node is constant, because it is chosen in the beginning, when the index is created and not modified at run-time. The input data for the R-tree is artificially generated so that each node has 96 disjoint child keys. On the GPU 128 scan tasks are scheduled at once as described in [7]. The machine used for the experiment is equipped with an Intel Xeon CPU operating at 2.3 GHz and an NVIDIA Tesla C1060 GPU. [4]

The sort workload is used to test our model for mutli-parameter estimations. The first parameter is the number of 32-bit integers contained in the array to be sorted and the second parameter the number of threads that are available on the CPU for processing. We assume that the GPU is always free for processing and can use all its cores. The machine used for this experiment is a Z600 workstation with two Intel Xeon E5-2630 with 6 cores each and the Tesla C2050. [5] Data transfer times are included in all of the measurements.

[4] See Appendix A.1 for Details on Hardware.
[5] See Appendix A.1 for Details on Hardware.



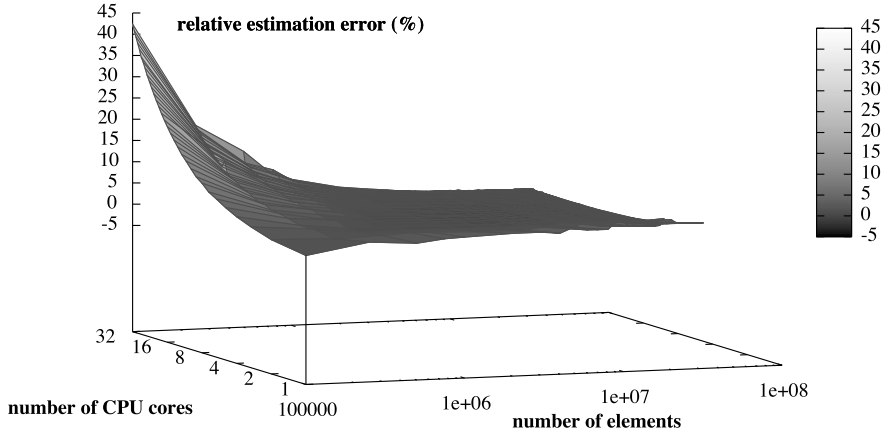




Figure 6.10.: Relative sorting estimation errors, training length: 300 operations

### 6.4.3. Model Validation

In this section we evaluate the scheduling decisions of the HyPE framework. First, we executed a training phase for the use case with input data generated from the entire parameter space. The framework creates a global execution model for the trained problem class. Second, we compare the scheduling decision made by HyPE based on estimated run-times with the real run-time.

With an increasing number of parameter dimensions it becomes unfeasible to train the framework with all parameter combinations. For the sort workload, we choose 50 parameter combinations randomly from the generated workload. Figure 6.9 shows the relative estimations errors for the whole parameter space. The relative estimation error denotes the average absolute difference between each estimation value and its corresponding measure in the real workload. After this initial training phase, we added 250 additional samples. The new relative estimation errors are plotted in Figure 6.10.

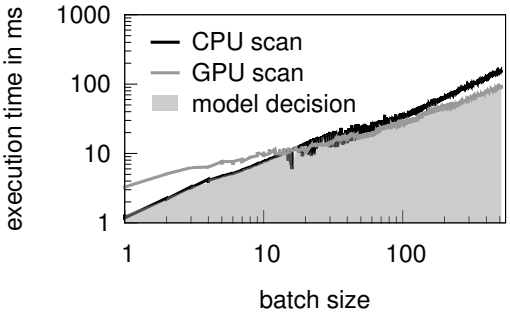



Figure 6.11.: Scheduling decisions for the index workload

As expected the largest errors occur in the area of many CPU cores and small data sizes. We already discussed that the execution times for this case are very unstable. Hence, a good prediction based on the given parameters is impossible. For most parameter combinations the estimation errors are $\approx 0$, which confirms that our model can effectively handle multiple parameters. The model becomes more accurate at run-time, when samples are added by executing the sort operation. In consequence scheduling decision for frequently executed operations are more accurate than for operation that are rarely executed.

The execution times for the index scan operation on CPU and GPU are illustrated in Figure 6.11. Because we used only one parameter, there is only one break-even point. After a short training phase, the relative estimation error for CPU scans is less than 9% and—since for this experiment again the GPU scan's run-time is more stable than the CPU's—the estimation error for the GPU scan is even smaller at around 4%. The shaded area shows the run-time of the respective model decision after the training. Together the HyPE and the GiST framework make sure that the application that uses the index gets the best performance without any knowledge about the underlying hardware.

### 6.4.4. Model Improvement

We showed that the run-time estimations of the HyPE framework are accurate and that in most cases the right algorithm is chosen. In this section we evaluate if we can benefit from hybrid processing in case of a typical workload. To quantify the performance gain we define a measure, the **model improvement** [12] as:

$$model\ improvement(DM_i \rightarrow DM_j, W) = \frac{T_{DM_i}(W) - T_{DM_j}(W)}{T_{DM_i}(W)} \tag{6.5}$$

The model improvement for a specific workload $W$ is a ratio that shows the impact on the run-time $T$ of a decision model $DM_i$ compared to another model $DM_j$. A positive value means that $DM_j$ allows the workload to be completed in a shorter run-time than $DM_i$, i.e., we benefit from using $DM_j$. Hence, not the sole number of wrong decisions is crucial for the model improvement but the decision's impact. Wrong scheduling decisions near the break-even-point(s) are uncritical for most users.

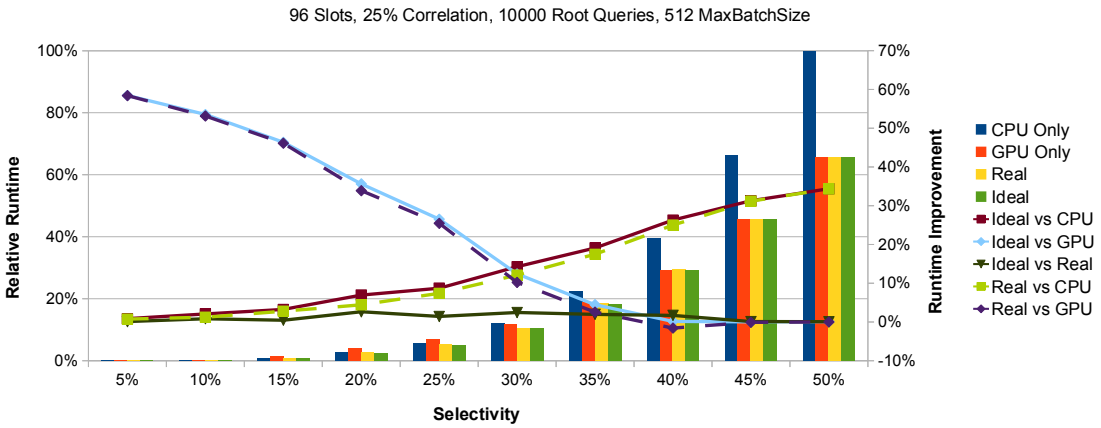



Figure 6.12.: Model improvement for the index scan depending on selectivity

We define $DM_{ideal}$ as the hypothetical model that always chooses the fastest algorithm and processing unit. It is an upper bound that cannot be achieved in practice because there is always overhead for making the decision and adapting the learned model. The goal is to get as close as possible to $DM_{ideal}$. A hybrid approach is beneficial when the model improvement is positive compared to the trivial models that always choose the same algorithm for the same processing unit. Otherwise, the overhead for learning and adapting parameters cancels out any performance gain.

To evaluate the model improvement of HyPE in combination with the GiST framework we create an R-Tree with equally structured nodes and non-overlapping keys. It has 5 levels and 96 slots, leading to a total number of 8 billion indexed entries. For every query workload we use predicates with a different *selectivity*. The higher the selectivity the more queries are sent to the child nodes, i.e., a higher selectivity means more queries per batch. Figure 6.12 shows the normalized total run-times for each decision model illustrated as bars. The differences between the run-times are depicted as curves. For selectivities below 20% the model that always chooses the CPU is almost ideal, because all batches except for the root-node queries are small. When the selectivity increases to over 40% batch sizes are large enough so that always choosing the GPU is the ideal model. If the selectivity is between 20% and 40% neither of the trivial models are ideal. HyPE however is close to the ideal model (within 5% including the overhead for adapting the model) in every case. Compared to the trivial models we achieve a significant performance improvement.

## 6.5. Related Work

**Analytical Cost Models** While we estimate execution times based on records of previous runs, there are other possibilities as well. Manegold et al. presented a framework that creates cost functions of database operations with the help of their memory access patterns. The cost functions are used to estimate the execution time of an operator [66].

Together with the query operators He et al. present a cost model that can be used to predict the run time of a query on the GPU in [42]. However, both models are limited to operators used in query execution, while we focus on a more general model. Kothapalli et al. propose to analyse CUDA kernel to predict the performance of any kernel [59].

**Learning based Execution Time Estimation** Similar to our approach, others also developed learning models, but used them in another context. For example, Zhang et al. used a learning model to predict the costs of XML queries. Akdere et al. proposed the Predictive DBMS in which the optimizer trains and selects models transparent to the user [3]. The basic idea of their approach is to perform a feature extraction on queries and compute execution time estimations based on them. Instead of predicting the execution time Matsunaga et al. presented an approach to estimate the resource usage for an application [68].

**Decision Models** Kerr et al. also worked on the possibility to automatically decide whether to use the GPU or not. Instead of a learning based approach they analyzed CUDA kernels and recorded a high number of metrics. These metrics are then used to decide on any given kernel written in CUDA. This choice is made statically in contrast to our work and introduces no run-time overhead but cannot adapt. Iverson et al. developed an approach that estimates execution times of tasks in the context of distributed systems [50]. The approach, similar to our model, does not require hardware-specific information, but differs in focus and statistical methods from ours.

## 6.6. Conclusions

In this chapter we presented a framework that schedules tasks to available processing units by using knowledge from previous executions. Our approach uses spline-interpolation to create a cost function for each operation and processing unit. Therefore, no detailed knowledge about the CPU or the co-processor is necessary, both are treated as black boxes. Every execution refines the cost function, and it can easily be re-calibrated in case the hardware changes. The evaluation results show that our approach achieves near optimal decisions and quickly adapts to workloads.

# 7. Conclusion

The first thing we learned while doing research on GPUs in DBMS was that although we can execute almost any task with the help of OpenCL or CUDA, they are not necessarily faster on the GPU. On the one hand, we have the transfer bottleneck. No matter, what technologies (UVA) or approaches (caching) we use, we always have to copy data to the device memory. On the other hand, not every algorithm can be fitted to the programming model used by the GPU. In general more work has to be done in parallel algorithms. Depending on the complexity of the partition and merge operations this overhead might outweigh the benefit of having more calculation power. Sometimes the overhead grows with the number of cores we want to use. Then the parallel algorithm scales well on a multi-core CPU, but fails on the GPU, where 100 times more threads are necessary.

Nevertheless, we could show that there are tasks within a DBMS that can benefit from the raw calculation power of the GPU. First, there is application logic; machine learning algorithms such as K-Means are known to run faster on the GPU. We showed that we can have the advantages of executing it in the DBMS as UDF and on the GPU at the same time. Second, query optimization depends on selectivity estimations, which have to be calculated with the help of linear algebra. GPU vendors provide libraries factors faster than comparable functions on the CPU. By using these libraries we can use more complex operations, so query execution can benefit from more accurate estimations. Third, although the GPU is not necessarily better at executing OLAP queries, there are queries where we are factor ten faster as long as the data fits into the GPU's memory. Using the GPU for stored procedures that are executed regularly can free resources on the CPU for other jobs and provide results faster.

It is unlikely that the system can estimate a task's run-time accurately by just analyzing the code, because the execution time strongly depends on data and query characteristics. These are hard to measure and quantify. Estimations for static tasks based on earlier executions, however, are a good basis for decisions. When implementations for the CPU and a co-processor are available, e.g., with the help of OpenCL, the system is able to choose the best processing unit and algorithm without user-interaction. The overhead for using a framework that provides this is minimal as we have shown in the previous chapter.

The GPU provides so much calculation power, because most of the transistors on it are actually used to process instructions. The downside of this approach is that the code has to be optimized to use the GPU in the right way. Therefore, unoptimized code has a heavy impact on the performance of the GPU. In contrast CPUs provide hardware to optimize the execution of the instructions, which compensates for such code. Writing good code requires constant training and knowledge about the architecture. So instead of programming for the GPU directly, most developers should use libraries for

execution, even if this requires to transform the actual problem to be solved by the given functionality. In most cases the performance loss because of transformations will be smaller than the loss because of poorly written code.

Former Co-processors, like the FPU or processors for cryptography are integrated into the CPU nowadays. To use these special function units, CPU vendors provide special instruction sets, which are automatically used by compilers. Hence, application developers usually do not have to care about where tasks are executed. While we also see clear signs that the GPU will be integrated into a heterogeneous processing platform as well, it is unlikely that we can use them transparently. Of course there will be special instructions for graphics processing, but in contrast to the FPU for instance, the GPU is able to solve problems from other domains as well. So, if performance is a priority, developers have to decide for each and every task, where to execute it. This problem will even be more challenging than today. There is no transfer bottleneck on heterogeneous platforms, but CPU and GPU share resources. Using the GPU might slow down the CPU, because the memory bus is used or because the temperature on the chip is too high for both to run at full power. Therefore, it is even more important to consider the right architecture for a problem. First, the algorithm itself has to be analyzed. On the one hand our dictionary merge example from Section 4.3 can be solved efficiently with one thread. The more threads we use, the more overhead is necessary. K-Means on the other hand requires a high number of distances to be calculated, which can be done independently and in parallel without much overhead. Second, if the algorithm fits to the parallel architecture of the GPU the input data has to be checked at run-time whether it allows the problem to be distributed over many cores or not. Only then it makes sense to use the GPU.

# Bibliography


[1] Daniel J. Abadi, Samuel R. Madden, and Miguel Ferreira. Integrating compression and execution in column-oriented database systems. In *Proc. of ACM SIGMOD Conference*, page 671. ACM Press, June 2006.

[2] Daniel J. Abadi, Samuel R. Madden, and Nabil Hachem. Column-Stores vs. Row-Stores: How Different Are They Really? In *Proc. of ACM SIGMOD Conference*, page 967, New York, New York, USA, 2008. ACM Press.

[3] Mert Akdere, Ugur Cetintemel, Matteo Riondato, Eli Upfal, and Stanley Zdonik. The Case for Predictive Database Systems: Opportunities and Challenges. In *Proc. of CIDR Conference*, 2011.

[4] ALGLIB Project. Homepage of ALGLIB. http://www.alglib.net/.

[5] Peter A. Alsberg. Space and time savings through large data base compression and dynamic restructuring. *IEEE*, 63:1114–1122, 1975.

[6] Peter Bakkum and Kevin Skadron. Accelerating SQL database operations on a GPU with CUDA. In *GPGPU Workshop*, page 94, New York, New York, USA, 2010. ACM Press.

[7] Felix Beier, Torsten Kilias, and Kai-Uwe Sattler. GiST scan acceleration using coprocessors. In *Proc. of DaMoN Workshop*, pages 63–69, New York, New York, USA, May 2012. ACM Press.

[8] Manfred Bertuch, Hartmut Gieselmann, Andrea Trinkwalder, and Christof Windeck. Supercomputer zu Hause. *c't Magazin*, 7, 2009.

[9] Carsten Binnig, Norman May, and Tobias Mindnich. SQLScript: Efficiently Analyzing Big Enterprise Data in SAP HANA. *BTW*, 2013.

[10] Peter Boncz, Marcin Zukowski, and Niels Nes. MonetDB/X100: Hyper-Pipelining Query Execution. In *Proc. of CIDR Conference*, volume 5, 2005.

[11] Hanan Boral and David J. DeWitt. Database machines: An idea whose time has passed? a critique of the future of database machines. Technical report, 1983.

[12] Sebastian Breß, Felix Beier, Hannes Rauhe, Kai-Uwe Sattler, Eike Schallehn, and Gunter Saake. Efficient Co-Processor Utilization in Database Query Processing. *Information Systems*, 2013.

[13] Sebastian Breß, Siba Mohammad, and Eike Schallehn. Self-Tuning Distribution of DB-Operations on Hybrid CPU/GPU Platforms. *Grundlagen von Datenbanken*, 2012.

[14] Nathan Brookwood. Whitepaper: AMD Fusion Family of APUs: Enabling a Superior, Immersive PC Experience, 2010.

[15] Donald D. Chamberlin. *A Complete Guide to DB2 Universal Database*. Morgan Kaufmann, 1998.

[16] George P. Copeland and Setrag N. Khoshafian. A decomposition storage model. In *ACM SIGMOD Record*, volume 14, pages 268–279. ACM, 1985.

[17] Mayank Daga, Ashwin Aji, and Wu-Chun Feng. On the Efficacy of a Fused CPU+GPU Processor (or APU) for Parallel Computing. In *SAAHPC*, pages 141–149. IEEE, July 2011.

[18] Jeffrey Dean and Sanjay Ghemawat. MapReduce. *Communications of the ACM*, 51(1):107, January 2008.

[19] Jonathan Dees and Peter Sanders. Efficient Many-Core Query Execution in Main Memory Column-Stores. In *Proc. of ICDE Conference*, 2013.

[20] Cristian Diaconu, Craig Freedman, Erik Ismert, and Per-Ake Larson. Hekaton: SQL Server's memory-optimized OLTP engine. In *Proc. of VLDB Conference*, 2013.

[21] Franz Färber, Sang Kyun Cha, Jürgen Primsch, and Christof Bornhövd. SAP HANA database: data management for modern business applications. In *Proc. of ACM SIGMOD Conference*, 2012.

[22] Franz Färber, Norman May, Wolfgang Lehner, Philipp Groß e, Ingo Müller, Hannes Rauhe, and Jonathan Dees. The SAP HANA Database – An Architecture Overview. *IEEE Data Eng. Bull.*, 35(1):28–33, 2012.

[23] Rob Farber. CUDA, Supercomputing for the Masses: Part 21. *Dr. Dobb's Journal* , November 2010.

[24] Michael J. Flynn. Some Computer Organizations and Their Effectiveness. *IEEE Transactions on Computers*, C-21(9):948–960, September 1972.

[25] Free Software Foundation. The GNU C++ Library - Online Dokumentation. http://gcc.gnu.org/onlinedocs/libstdc++/index.html.

[26] Hector Garcia-Molina and Kahne Salem. Main memory database systems: an overview. *IEEE Transactions on Knowledge and Data Engineering*, 4(6):509–516, 1992.

[27] Bura Gedik, Rajesh R. Bordawekar, and Philip S. Yu. CellSort: high performance sorting on the cell processor. pages 1286–1297, September 2007.

[28] Bura Gedik, Rajesh R. Bordawekar, and Philip S. Yu. CellJoin: a parallel stream join operator for the cell processor. *The VLDB Journal*, 18(2):501–519, September 2008.

[29] Bura Gedik, Philip S. Yu, and Rajesh R. Bordawekar. Executing stream joins on the cell processor. pages 363–374, September 2007.

[30] Jongmin Gim and Youjip Won. Extract and infer quickly: Obtaining sector geometry of modern hard disk drives. *ACM Transactions on Storage*, 6(2):1–26, July 2010.

[31] Brian Gold, Anastasia Ailamaki, Larry Huston, and Babak Falsafi. Accelerating database operators using a network processor. *Proc. of DaMoN Workshop*, page 1, 2005.

[32] Solomon W. Golomb. Run-length encodings. *IEEE Trans Info Theory 12(3)*, 1966.

[33] Naga Govindaraju, Jim Gray, Ritesh Kumar, and Dinesh Manocha. GPUTeraSort. In *Proc. of ACM SIGMOD Conference*, page 325, New York, New York, USA, June 2006. ACM Press.

[34] Naga Govindaraju, Brandon Lloyd, Wei Wang, Ming Lin, and Dinesh Manocha. Fast computation of database operations using graphics processors. In *Proc. of ACM SIGMOD Conference*, pages 215–226. ACM, 2004.

[35] Goetz Graefe. Encapsulation of parallelism in the Volcano query processing system. In *Proc. of ACM SIGMOD Conference*, volume 19, pages 102–111, New York, New York, USA, May 1990. ACM Press.

[36] Goetz Graefe. Volcano—An Extensible and Parallel Query Evaluation System. *IEEE Transactions on Knowledge and Data Engineering*, 6(1):120–135, 1994.

[37] Chris Gregg and Kim Hazelwood. Where is the data? Why you cannot debate CPU vs. GPU performance without the answer. In *IEEE International Symposium on Performance Analysis of Systems and Software*, pages 134–144. IEEE, April 2011.

[38] P. Griffiths Selinger, Morton M. Astrahan, Donald D. Chamberlin, Raymond A. Lorie, and Thomas G. Price. Access path selection in a relational database management system. In *Proc. of ACM SIGMOD Conference*, page 23, New York, New York, USA, May 1979. ACM Press.

[39] Philipp Groß e, Wolfgang Lehner, Thomas Weichert, Franz Färber, and Wen-Syan Li. Bridging two worlds with RICE—Integrating R into the SAP In-Memory Computing Engine. *Proc. of VLDB Conference*, 4(12), 2011.

[40] Silviu Guiasu and Abe Shenitzer. The principle of maximum entropy. *The Mathematical Intelligencer*, 7(1):42–48, March 1985.

[41] Theo Haerder and Andreas Reuter. Principles of transaction-oriented database recovery. *ACM Computing Surveys*, 15(4):287–317, December 1983.

[42] Bingsheng He, Mian Lu, Ke Yang, Rui Fang, Naga Govindaraju, Qiong Luo, and Pedro V. Sander. Relational query coprocessing on graphics processors. *ACM Transactions on Database Systems*, 34(4):1–39, December 2009.

[43] Bingsheng He, Ke Yang, Rui Fang, Mian Lu, Naga Govindaraju, Qiong Luo, and Pedro V. Sander. Relational joins on graphics processors. *Proc. of ACM SIGMOD Conference*, page 511, 2008.

[44] Bingsheng He and Jeffrey Xu Yu. High-Throughput Transaction Executions on Graphics Processors. In *Proc. of VLDB Conference*, volume 4, pages 314–325, March 2011.

[45] Max Heimel and Volker Markl. A first step towards gpu-assisted query optimization. In *Proc. of ADMS workshop*, 2012.

[46] Max Heimel, Michael Saecker, Holger Pirk, Stefan Manegold, and Volker Markl. Hardware-oblivious parallelism for in-memory column-stores. In *Proc. of VLDB Conference*, volume 6, pages 709–720. VLDB Endowment, July 2013.

[47] Joseph M. Hellerstein, Jeffrey F. Naughton, and Avi Pfeffer. Generalized Search Trees for Database Systems. In *Proc. of VLDB Conference*, pages 562–573. Morgan Kaufmann Publishers Inc., September 1995.

[48] Wen-mei W. Hwu. *GPU Computing Gems Jade Edition*. Morgan Kaufmann Publishers Inc., October 2011.

[49] Intel. Intel microprocessor export compliance metrics. http://www.intel.com/support/processors/sb/CS-017346.htm.

[50] Michael A. Iverson, Fusun Ozguner, and Gregory J. Follen. Run-Time Statistical Estimation of Task Execution Times for Heterogeneous Distributed Computing. In *Proc. of HPDC Symposium*, page 263. IEEE Computer Society, August 1996.

[51] Jens Steube. Homepage of oclHashcat. http://hashcat.net/oclhashcat/.

[52] Tim Kaldewey, Guy M. Lohman, Rene Mueller, and Peter Volk. GPU Join Processing Revisited. *Proc. of DaMoN Workshop*, 2012.

[53] Kamran Karimi, Neil G. Dickson, and Firas Hamze. A Performance Comparison of CUDA and OpenCL. page 12, May 2010.

[54] Tomas Karnagel, Dirk Habich, Benjamin Schlegel, and Wolfgang Lehner. The HELLS-join. In *Proc. of DaMoN Workshop*, page 1, New York, New York, USA, June 2013. ACM Press.

[55] Martin Kaufmann, Amin Amiri Manjili, Panagiotis Vagenas, Peter Michael Fischer, Donald Kossmann, Franz Färber, and Norman May. Timeline index. In *Proc. of ACM SIGMOD Conference*, page 1173, New York, New York, USA, June 2013. ACM Press.

[56] Alfons Kemper and Thomas Neumann. HyPer: A hybrid OLTP&OLAP main memory database system based on virtual memory snapshots. In *Proc. of ICDE Conference*, pages 195–206. IEEE, April 2011.

[57] Laszlo B. Kish. End of Moore's law: Thermal (noise) death of integration in micro and nano electronics. *Physics Letter A*, 305:144–149, 2002.

[58] Veit Köppen, Gunter Saake, and Kai-Uwe Sattler. *Data Warehouse Technologien*. mitp-Verlag, 2012.

[59] Kishore Kothapalli, Rishabh Mukherjee, Suhail Rehman, Suryakant Patidar, Punjab J. Narayanan, and Kannan Srinathan. A performance prediction model for the CUDA GPGPU platform. In *High Performance Computing Conference (HiPC)*, pages 463–472. Ieee, 2009.

[60] Konstantinos Krikellas, Stratis D. Viglas, and Marcelo Cintra. Generating code for holistic query evaluation. In *Proc. of ICDE Conference*, pages 613–624. IEEE, March 2010.

[61] Jens Krueger, Martin Grund, Ingo Jaeckel, Alexander Zeier, and Hasso Plattner. Applicability of GPU Computing for Efficient Merge in In-Memory Databases. In *Proc. of ADMS Workshop*, 2011.

[62] Victor W. Lee, Per Hammarlund, Ronak Singhal, Pradeep Dubey, Changkyu Kim, Jatin Chhugani, Michael Deisher, Daehyun Kim, Anthony D. Nguyen, Nadathur Satish, Mikhail Smelyanskiy, and Srinivas Chennupaty. Debunking the 100X GPU vs. CPU myth. In *Proc. of ISCA*, page 451, New York, New York, USA, 2010. ACM Press.

[63] Christian Lemke, Kai-Uwe Sattler, and Franz Färber. Compression Techniques for Column-Oriented BI Accelerator Solutions. In *BTW*, 2009.

[64] David Levinthal. Performance Analysis Guide for Intel Core i7 Processor and Intel Xeon 5500 processors. Technical report, 2009.

[65] James B. Macqueen. Some Methods for classification and analysis of multivariate observations. 1:281 – 297, 1967.

[66] Stefan Manegold, Peter Boncz, and Martin L. Kersten. Generic database cost models for hierarchical memory systems. In *Proc. of VLDB Conference*, pages 191–202. VLDB Endowment, August 2002.

[67] Volker Markl, Peter J. Haas, Marcel Kutsch, Nimrod Megiddo, Utkarsh Srivastava, and Tam M. Tran. Consistent selectivity estimation via maximum entropy. *The VLDB Journal*, 16(1):55–76, September 2006.

[68] Andréa Matsunaga and José A.B. Fortes. On the Use of Machine Learning to Predict the Time and Resources Consumed by Applications. In *Proc. of CCGRID Conference*, pages 495–504. IEEE, May 2010.

[69] Gordon E. Moore. Cramming More Components Onto Integrated Circuits. *Electronics*, pages 114–117, January 1965.

[70] Rene Mueller and Jens Teubner. FPGAs: A New Point in the Database Design Space. In *Proc. of EDBT Conference*, page 721. ACM Press, March 2010.

[71] Nadathur Satish, Changkyu Kim, Jatin Chhugani, Anthony D. Nguyen, Victor W. Lee, Daehyun Kim, and Pradeep Dubey. Fast Sort on CPUs, GPUs and Intel MIC Architectures.

[72] Thomas Neumann. Efficiently compiling efficient query plans for modern hardware. In *Proc. of VLDB Conference*, volume 4, pages 539–550, 2011.

[73] NVIDIA. CUDA C Best Practices Guide. http://docs.nvidia.com/cuda/pdf/CUDA_C_Best_Practices_Guide.pdf.

[74] NVIDIA. Homepage of the CUDA SDK Documentation. http://docs.nvidia.com/cuda/index.html.

[75] NVIDIA. Homepage of Thrust. https://developer.nvidia.com/Thrust.

[76] NVIDIA. NVIDIA Tesla Kepler Datasheet. http://www.nvidia.com/content/tesla/pdf/NVIDIA-Tesla-Kepler-Family-Datasheet.pdf.

[77] NVIDIA. Whitepaper: NVIDIAs Next Generation CUDATMCompute Architecture: Kepler GK110. Technical report, 2012.

[78] Pat O'Neil, Betty O'Neil, and Xuedong Chen. Star Schema Benchmark, 5 June 2009.

[79] OpenACC. Homepage of OpenACC. http://www.openacc-standard.org/.

[80] Oracle. Whitepaper: Extreme Performance Using Oracle TimesTen In-Memory Database. Technical report, 2009.

[81] Carlos Ordonez. Programming the K-means clustering algorithm in SQL. In *Proc. of ACM SIGKDD*, page 823, New York, New York, USA, August 2004. ACM Press.

[82] John D. Owens, David Luebke, Naga Govindaraju, Mark Harris, Jens Krüger, Aaron E. Lefohn, and Timothy J. Purcell. A Survey of GeneralPurpose Computation on Graphics Hardware. *Computer Graphics Forum*, 26:80–113, 2007.

[83] Hasso Plattner and Alexander Zeier. *In-Memory Data Management: An Inflection Point for Enterprise Applications*. 2011.

[84] Viswanath Poosala and Yannis E. Ioannidis. Selectivity Estimation Without the Attribute Value Independence Assumption. In *Proc. of VLDB Conference*, pages 486–495. Morgan Kaufmann Publishers Inc., August 1997.

[85] Iraklis Psaroudakis, Tobias Scheuer, Norman May, and Anastasia Ailamaki. Task Scheduling for Highly Concurrent Analytical and Transactional Main-Memory Workloads. In *Proc. of ADMS Workshop*, 2013.

[86] Vijayshankar Raman, Gopi Attaluri, Ronald Barber, Naresh Chainani, David Kalmuk, Vincent KulandaiSamy, Jens Leenstra, Sam Lightstone, Shaorong Liu, Guy M. Lohman, Tim Malkemus, Rene Mueller, Ippokratis Pandis, Berni Schiefer, David Sharpe, Richard Sidle, Adam Storm, and Liping Zhang. DB2 with BLU acceleration: so much more than just a column store. In *Proc. of VLDB Conference*, volume 6, pages 1080–1091. VLDB Endowment, August 2013.

[87] Cornelius Ratsch. Adaptive String Dictionary Compression in In-Memory Column-Store Database Systems, 2013.

[88] Hannes Rauhe, Jonathan Dees, Kai-Uwe Sattler, and Franz Färber. Multi-level Parallel Query Execution Framework for CPU and GPU. *Proc. of ADBIS Conference*, 2013.

[89] Michael Rudolf, Marcus Paradies, Christof Bornhövd, and Wolfgang Lehner. SynopSys: Large Graph Analytics in the SAP HANA Database Through Summarization. In *Proc. of GRADES Workshop*, 2013.

[90] Michael Rudolf, Marcus Paradies, Christof Bornhövd, and Wolfgang Lehner. The Graph Story of the SAP HANA Database. *BTW*, 2013.

[91] Larry Seiler, Robert Cavin, Roger Espasa, Ed Grochowski, Toni Juan, Pat Hanrahan, Doug Carmean, Eric Sprangle, Tom Forsyth, Michael Abrash, Pradeep Dubey, Stephen Junkins, Adam Lake, and Jeremy Sugerman. Larrabee: a many-core x86 architecture for visual computing. In *Proc. of ACM SIGGRAPH Conference*, volume 27, page 1. ACM Press, August 2008.

[92] Vishal Sikka, Franz Färber, Wolfgang Lehner, and Sang Kyun Cha. Efficient transaction processing in SAP HANA database: the end of a column store myth. *Proceedings of the ACM SIGMOD*, 2012.

[93] Steve Rennich (NVIDIA). Streams And Concurrency Webinar. http://developer.download.nvidia.com/CUDA/training/StreamsAndConcurrencyWebinar.pdf, 2012.

[94] Mike Stonebraker, Daniel J. Abadi, Adam Batkin, Xuedong Chen, Mitch Cherniack, Miguel Ferreira, Edmond Lau, Amerson Lin, Samuel R. Madden, and Elizabeth O'Neil. C-Store: A Column-oriented DBMS. In *Proc. of VLDB Conference*, pages 553–564. VLDB Endowment, 2005.

[95] Stuart P. Lloyd. Least squares quantization in PCM. *Bell Laboratories Memorandum*, 1957. also published in IEEE Trans. Inform. Theory, vol. IT-28, pp. 129-137.

[96] George Teodoro, Tahsin Kurc, Jun Kong, Lee Cooper, and Joel Saltz. Comparative Performance Analysis of Intel Xeon Phi, GPU, and CPU, November 2013.

[97] Jens Teubner and Rene Mueller. How soccer players would do stream joins. In *Proc. of ACM SIGMOD Conference*, page 625, New York, New York, USA, June 2011. ACM Press.

[98] The C-Store Project Group. Homepage of C-Store. https://db.csail.mit.edu/projects/cstore/.

[99] The clang community. Homepage of clang: a C language family frontend for LLVM. http://clang.llvm.org/.

[100] Transaction Processing Performance Council. Homepage of the TPC-H Benchmark. http://www.tpc.org/tpch/.

[101] Leslie G. Valiant. A Bridging Model for Parallel Computation. *Communications of the ACM*, 33(8):103–111, August 1990.

[102] Panos Vassiliadis, Alkis Simitsis, and Spiros Skiadopoulos. Conceptual modeling for ETL processes. In *Proc. of DOLAP Workshop*, pages 14–21, New York, New York, USA, November 2002. ACM Press.

[103] Thomas Willhalm, Ismail Oukid, Ingo Mueller, and Franz Färber. Vectorizing Database Column Scans with Complex Predicates. In *Proc. of ADMS Workshop*, 2013.

[104] Thomas Willhalm, Nicolae Popovici, Yazan Boshmaf, Hasso Plattner, Alexander Zeier, and Jan Schaffner. SIMD-Scan: Ultra Fast in-Memory Table Scan using on-Chip Vector Processing Units. In *Proc. of VLDB Conference*, pages 385–394, Lyon, France, August 2009. ACM.

[105] Guanying Wu and Xubin He. Reducing SSD read latency via NAND flash program and erase suspension. In *USENIX Conference on File and Storage Technologies*, page 10. USENIX Association, February 2012.

[106] Yuan Yuan, Rubao Lee, and Xiaodong Zhang. The Yin and Yang of processing data warehousing queries on GPU devices. In *Proc. of VLDB Conference*, pages 817–828. VLDB Endowment, August 2013.

[107] Ning Zhang, Peter J. Haas, Vanja Josifovski, Guy M. Lohman, and Chun Zhang. Statistical learning techniques for costing XML queries. In *Proc. of VLDB Conference*, pages 289–300. VLDB Endowment, August 2005.

[108] Marcin Zukowski and Peter Boncz. MonetDB/X100-A DBMS In The CPU Cache. *IEEE Data Eng. Bull.*, 2005.

[109] Marcin Zukowski, Sandor Heman, Niels Nes, and Peter Boncz. Super-Scalar RAM-CPU Cache Compression. In *Proc. of ICDE Conference*, pages 59–59. IEEE, April 2006.

# A. Appendix

## A.1. Hardware Used for Evaluations

The experiments for this thesis were conducted on different machines. The machines with their CPUs and GPUs are listed here (bandwidth given in the form: *from device to host / host to device*):

| Machine alias | CPU | GPU | Bandwidth pageable | Bandwidth pinned |
|---|---|---|---|---|
| Tesla C1060 | 2 x E5520 | Tesla C1060 | 5.2 / 4.1 GB/s | n.a. |
| Z600 | 2 x X5650 | Tesla C2050 | 2.5 / 3.2 GB/s | 7.4 / 8.0 GB/s |
| K10 | 2 x E5-2690 | Tesla K10 | 2.3 / 2.8 GB/s | 11.2 / 12.2 GB/s |
| K20 | 2 x E5-2665 | Tesla K20m | 1.6 / 2.0 GB/s | 5.7 / 6.4 GB/s |

The bandwidth has been measured with the bandwidth test tool that is part of the CUDA toolkit. The value is not only depending on the GPU but also on the memory bus and the PCIe bus. Therefore,the same graphics card might behave differently in another system. In general we tried to use the newest available hardware for our experiments. In case of the string processing experiment, we waited for the K10 machine to be available, because it is the only machine with a working PCIe 3 connection and we were interested in fast transfers. The following table lists which machine was used for which experiment.

| Section | Experiment | Tesla C1060 | Z600 | K10 | K20 |
|---|---|---|---|---|---|
| 3.4.1 | Kernel Overhead/Bandwidth | | | | X |
| 3.4.2 | Single Core | | | | X |
| 3.4.3 | Streaming | | | | X |
| 3.4.4 | Matrix Multiplication | | | | X |
| 3.4.5 | String Processing | | | X | |
| 4.1.2 | K-Means UDF | | X | | |
| 4.2.3 | Maximum Entropy | | | | X |
| 4.3 | Dictionary Merge | | X | | |
| 5.5 | Query Execution | | X | | |
| 6.4 | Scheduling/Index Scan | X | | | |
| 6.4 | Scheduling/Sort | | X | | |

# Erklärung

Ich versichere, dass ich die vorliegende Arbeit ohne unzulässige Hilfe Dritter und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe. Die aus anderen Quellen direkt oder indirekt übernommenen Daten und Konzepte sind unter Angabe der Quelle gekennzeichnet.

An der inhaltlich-materiellen Erstellung der vorliegenden Arbeit waren keine weiteren Personen beteiligt. Insbesondere habe ich hierfür nicht die entgeltliche Hilfe von Vermittlungs- bzw. Beratungsdiensten (Promotionsberater oder anderer Personen) in Anspruch genommen. Niemand hat von mir unmittelbar oder mittelbar geldwerte Leistungen für Arbeiten erhalten, die im Zusammenhang mit dem Inhalt der vorgelegten Dissertation stehen.

Die Arbeit wurde bisher weder im In- noch im Ausland in gleicher oder ähnlicher Form einer Prüfungsbehörde vorgelegt.

Ich bin darauf hingewiesen worden, dass die Unrichtigkeit der vorstehenden Erklärung als Täuschungsversuch bewertet wird und gemäß §7 Abs. 10 der Promotionsordnung den Abbruch des Promotionsverfahrens zur Folge hat.

(Ort, Datum) (Unterschrift)