Datenbank Spektrum (2014) 14:199–209
DOI 10.1007/s13222-014-0164-z

SCHWERPUNKTBEITRAG

The Design and Implementation of CoGaDB: A Column-oriented GPU-accelerated DBMS

Sebastian Breß

Received: 26 May 2014 / Accepted: 30 July 2014 / Published online: 21 August 2014

Abstract Nowadays, the performance of processors is primarily bound by a fixed energy budget, the *power wall*. This forces hardware vendors to optimize processors for specific tasks, which leads to an increasingly heterogeneous hardware landscape. Although efficient algorithms for modern processors such as GPUs are heavily investigated, we also need to prepare the database optimizer to handle computations on heterogeneous processors. GPUs are an interesting base for case studies, because they already offer many difficulties we will face tomorrow.

In this paper, we present CoGaDB, a main-memory DBMS with built-in GPU acceleration, which is optimized for OLAP workloads. CoGaDB uses the self-tuning optimizer framework HyPE to build a hardware-oblivious optimizer, which learns cost models for database operators and efficiently distributes a workload on available processors. Furthermore, CoGaDB implements efficient algorithms on CPU and GPU and efficiently supports star joins. We show in this paper, how these novel techniques interact with each other in a single system. Our evaluation shows that CoGaDB quickly adapts to the underlying hardware by increasing the accuracy of its cost models at runtime.

Keywords DBMS architecture · GPU acceleration · Co-processing · Main-memory · DBMS

S. Breß (✉)
TU Dortmund University, 44227 Dortmund, Germany
e-mail: sebastian.bress@cs.tu-dortmund.de

S. Breß
University of Magdeburg, 39106 Magdeburg, Germany

1 Introduction

For the past years, modern processors are no longer primarily bound by transistor density but by a fixed energy budget, the *power wall* [8]. Thus, hardware vendors are forced to create specialized processing devices, which are optimized for a certain application field to further increase the performance of applications. Therefore, the hardware landscape becomes more heterogeneous, a trend that is expected to continue in the future [8]. We can already observe this trend today: Every desktop machine has a *Graphics Processing Unit* (GPU), a specialized co-processor for compute-intensive rendering tasks, which is also capable of general purpose processing. Therefore, GPUs are the pioneers of co-processing, followed by many alternative processor designs, such as *Multiple Integrated Cores* (MICs) or *Field Programmable Gate Arrays* (FPGAs).

In order to prepare database systems for tomorrows heterogeneous hardware landscape, many approaches were developed to efficiently execute database operators on GPUs. However, existing approaches and database systems making use of GPU acceleration have a common problem: They need to decide for each database operator, on which heterogeneous processor it should be executed. Typically, each system has a set of analytical cost models, which model the performance behavior of an operator on a particular processor (e.g., CPUs [25] or GPUs [17]). With an increasingly heterogeneous hardware landscape, this approach becomes more and more complex. Additionally, the database optimizer needs to take into account query plans with different operator placements, which increases the size of the optimization space. Some heterogeneous processors have their own dedicated device memory, which requires manual data placement and careful consideration, under which circumstances

data transfers will pay off. Since GPUs already offer these difficulties today, we argue that they are a suitable platform to study the problems every database system will face in the future.

However, we need to develop solutions that will scale also for future hardware. Thus, we need to address three basic problems:

1. Development of efficient algorithms for a set of heterogeneous processors.
2. Estimating the cost to execute a certain database operator on a certain processor, without detailed knowledge of the underlying hardware.
3. Efficiently balance a workload between all processors, where each heterogeneous processor performs the task it is most suited for.

In this paper, we present our system CoGaDB[1], a column-oriented, GPU-accelerated DBMS, which puts together our existing work in a high-performance OLAP engine that makes efficient use of GPUs to accelerate analytical query processing. Furthermore, we contribute a discussion of our design decisions, provide insights in CoGaDB's parallel query processor and CoGaDB's *Hybrid Query Processing Engine* (HyPE), which learns cost models for heterogeneous processor environments and optimizes query plans under consideration of special properties of GPUs. Thus, CoGaDB is the first DBMS that addresses problem 2 and problem 3. Problem 1 is the main focus of another system: Ocelot [19].

The remainder of the paper is structured as follows. We discuss background information on main-memory DBMSs and GPUs in Sect. 2. In Sect. 3, we will provide an overview of CoGaDB and discuss implementation details of the query processor in Sect. 4. Then, in Sect. 5, we present our optimizer HyPE, followed by a performance evaluation in Sect. 6. Afterwards, we outline our future development in Sect. 7, present related work in Sect. 8, and conclude in Sect. 9.

2 Background

In this section, we provide background information on main-memory DBMSs and GPUs.

2.1 Main-Memory DBMSs

With increasing capacity of main memory, it is possible to keep a large fraction of a database in memory. Thus, the performance bottleneck shifts from disk access to main-memory access, the *memory wall* [24]. For main-memory DBMSs, the architecture was heavily revised from a tuple-at-a-time volcano-style query processor to operator-at-a-time bulk processing and from a row-oriented data layout to columnar storage.[2] This increases the useful work per CPU cycle [7] and makes more efficient use of caches [24]. Furthermore, most systems compress data using light-weight compression techniques (e.g., dictionary encoding) to reduce the data volume and the required memory bandwidth for an operator [3].

2.2 GPUs

GPUs are specialized co-processors that are optimized for compute-intensive tasks. Thus, they spend most of their transistors on light-weight cores to allow for massive parallelism. Additionally, multiple cores are grouped in symmetric multiprocessors, which share the same instruction decoder. Thus, each multiprocessor natively implements the *Single Instruction Multiple Data* (SIMD) approach. In order to provide enough memory bandwidth to the multiprocessors, GPUs typically have a dedicated, high-bandwidth memory, which reduces the impact of the memory wall on the system.[3] However, GPUs with dedicated memory have a relatively small capacity compared to the CPUs main memory to keep reasonable monetary costs for a GPU. Thus, we can not cache the entire database in the GPUs memory but have to rely on a data placement strategy to transfer data between CPUs and GPUs. Data placement is especially important, because data transfers between CPU and GPU is the major performance bottleneck [16].

3 CoGaDB: A GPU-accelerated DBMS

In this section, we provide an overview of CoGaDB's architecture, including details on storage manager, processing and operator model, and GPU memory management. We illustrate CoGaDB's architecture in Fig. 1.

3.1 System Overview

CoGaDB's primary goal is to proof that we can optimize queries for heterogeneous processor machines without knowing the details of database algorithms and processors. Thus, we can build a query optimizer that scales with the increasing number of heterogeneous processors in today's and future machines without increasing maintenance effort. Since GPU acceleration is most beneficial for warehousing workloads, we designed CoGaDB as relational GPU-accelerated OLAP

[1] http://wwwiti.cs.uni-magdeburg.de/iti_db/research/gpu/cogadb/

[2] Many main-memory OLTP systems use a row-oriented data layout.

[3] We are aware of *Accelerated Processing Units* (APUs) from AMD, which integrate a CPU and a GPU on a single chip. However, APUs increase only the raw processing power of the machine, not memory bandwidth.

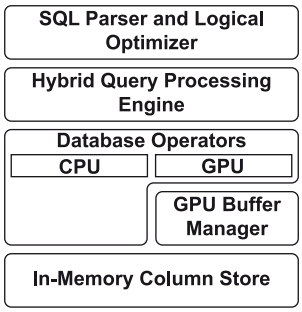

Fig. 1 CoGaDB's Architecture

engine. In order to quickly build a working prototype, we support currently only three data types: 32-Bit integers, 32-Bit floats, and variable-length strings. However, these data types are already sufficient to support the *Star Schema Benchmark* (SSBM), a popular OLAP benchmark [38].

In order to provide efficient data processing capabilities, CoGaDB makes use of parallel libraries, such as Intel's *Threading Building Blocks* (TBB) library[4] on the CPU and the Thrust library[5] on the GPU. As GPGPU framework, we decided to use NVIDIA's CUDA framework, because it has the most mature development toolkit and we expect a vendor-specific framework to deliver the best performance.

3.2 Storage Manager

As in every DBMS, the backbone of CoGaDB is its storage manager. Since we primarily target OLAP workloads, we choose a columnar data layout, because storing data column-wise allows for more efficient use of the memory hierarchy and higher compression rates. The work of He and others showed that GPU acceleration is not beneficial in case we have to fetch the data from disk [17]. Thus, another important consideration is that GPU acceleration is only beneficial in case we have all required data to answer a query cached in main memory. Therefore, CoGaDB's storage manager is an in-memory column store. In case the database fits not entirely into main memory, CoGaDB relies on the operating systems virtual memory management to swap cold data to disk.

3.3 Processing and Operator Model

As already mentioned, CoGaDB should be able to use all available processors to improve the performance of query processing. Therefore, we need a work unit on which we can build our operator placement. At the query level, a single processor is generally not suited for all operations contained in a query. Thus, we choose the operator level as granularity, because we can place operators to a processor suitable for that operator type.

There are basically two ways how to process queries: Pipelining, where each operator requests the next block of rows and bulk processing, where each operator consumes its input and materializes its output. In a heterogeneous processor machine, where we want to avoid data transfers between processors, we need to ensure that we assign each processor enough work in order to use the system efficiently (intra-operator parallelism). Furthermore, we can achieve parallelism between operators (inter-operator parallelism) if we construct bushy query plans and process independent sub-plans in parallel. In summary, CoGaDB uses the operator-at-a-time processing model and combines it with operator-based scheduling to distribute a set of queries on all available processing resources.

3.4 GPU Memory Management

A processor with dedicated memory requires a data placement strategy that moves data to the memory where it is needed. In CoGaDB, this is handled by the central GPU buffer manager. All GPU operators request their input columns from the buffer manager. If a column is not dormant in the GPUs memory, it is transferred to the GPU. The same principle is used for additional access structures, such as join indexes.

3.4.1 Memory Allocation Policy

Each GPU operator needs additional memory for data processing, such as memory for the result buffer and temporary data structures. There are two basic memory allocation strategies GPU operators can use. First, allocate the complete memory it needs to complete the computation (pre-allocation). Second, allocate memory as late as possible (allocate as needed). The first strategy avoids GPU operator abortions due to out-of-memory conditions during processing, whereas the allocate-as-needed strategy uses the GPU memory more economical, allowing for concurrently executed GPU operators or a larger GPU buffer for recently used columns. Additionally, the pre-allocation strategy is hard to implement because it is difficult to accurately estimate the result size of an operator. Thus, CoGaDB uses an allocate-as-needed strategy for memory allocation.

3.4.2 Memory Deallocation Policy

Since the memory capacity of a GPU is limited compared to the CPUs memory, it is likely that a GPU operator using the allocate-as-needed strategy will run out of memory while processing an operator, especially in workloads with parallel queries. In this case, cached data has to be removed from

[4] https://www.threadingbuildingblocks.org.
[5] http://thrust.github.io.

the GPU memory. CoGaDB first removes cached columns, because it is relatively cheap to copy them again to the GPU compared to join indexes. In case the removal of columns did not free enough memory, the cached join indexes are removed from the GPU memory.

3.4.3 Fault Tolerance

In case the GPU operator still has insufficient GPU memory after the memory cleanup, it has two options. First, it can wait until enough memory is available for allocation. Second, it can abort, discarding the work it has done so far and starting a fall-back handler that processes the operator on the CPU. With the first option, however, we can run into a similar situation as two transactions waiting for two locks: A cyclic dependency may occur, which causes a deadlock. Consider two GPU operators O_1 and O_2. Operator O_1 and O_2 allocate memory for their first processing step. Then, for a second processing step, both operators try to allocate more memory, but both fail, because O_1 and O_2 have already allocated GPU memory in the first processing step.

Thus, CoGaDB uses the second strategy, aborting a GPU operator and restarting the operator on the CPU. This strategy is similar to timestamp ordering of transactions, because the operator that comes to late (runs out of memory) is aborted. However, depending on the workload, the costs due to operator abortions can be significant.

4 Query Processor

In this section, we present details on CoGaDB's query processor, including operators, materialization strategy, operator parallelism, and parallel star joins.

4.1 Operators

We now elaborate details on the relational operators implemented in CoGaDB.

4.1.1 Selection

On the CPU, we use a parallel version of the predicated SIMD scan from Zhou and Ross [40]. Each thread scans its own partition of the input column and writes its matching TIDs to a local output buffer. However, CoGaDB requires that selection operators return a sorted continuous array of TIDs, so we need to combine the local results. Since each thread counts its number of matches, we can use this information to compute the size of the complete result and the memory region, where each thread has to copy their results into. The last step is also done in parallel.

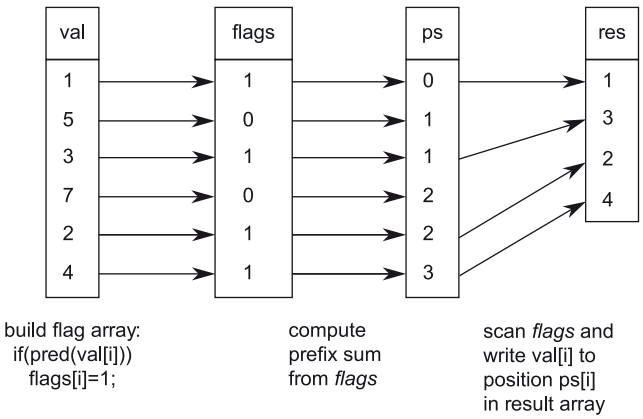

Fig. 2 Parallel selection on GPUs for predicate $val < 5$

On the GPU, CoGaDB uses the parallel selection of He and others [17], which we illustrate in Fig. 2. First, the operator performs a first scan of the column to compute a flag array, where a flag is one, if and only if the predicate matched the row. Second, a prefix sum is computed from the flag array to obtain the result size and the write positions of every thread in the output buffer. Finally, the input column is read again but this time each thread knows the position where it has to write its result.

CoGaDB also supports string columns on the GPU by applying dictionary compression, which allows CoGaDB to work on compressed values. Since the dictionary is not sorted, we can currently evaluate only is-equal and is-unequal predicates.

4.1.2 Complex Selection

Many real world queries do not filter one column only but define complex predicates.

Heimel and others identify three strategies for evaluating complex predicates [20]: chaining operators, complex predicate interpretation, and dynamic code generation and compilation. The chaining operators strategy builds a query plan where each predicate is evaluated independently, followed by operators that merge the result (e.g., combining bitmaps or lists of TIDs). Complex predicate interpretation stores the predicates in an abstract syntax tree and evaluates all predicates directly with a single pass over a table. Dynamic code generation and compilation creates a kernel that directly evaluates all predicates. Similar to complex predicate interpretation, the query compilation technique needs only a single pass over the data. However, query compilation suffers from high upfront costs of compiling a kernel, whereas complex predicate interpretation requires a large kernel with several conditional statements, which leads to branch divergence and, hence, inefficiency on GPUs.

In contrast, the chaining operators strategy allows CoGaDB to use very efficient GPU kernels to evaluate single predicates and to perform the result combination. Scans on different columns can be evaluated on separate processors in parallel to hide the costs of multiple passes over the data. Since CoGaDB's selection operators return sorted lists of TIDs, we use union and intersection operators to compute disjunctions and conjunctions, respectively. We use as set operators the parallel GPU algorithms of the Thrust library.

4.1.3 Join

The most time-intensive operator during relational OLAP is the computation of joins between the fact table and a dimension table. Thus, it is crucial to support efficient join implementations. CoGaDB offers three different join types: a generic join, a primary-key/foreign-key join (PK-FK join), and a fetch join. The generic join makes no assumptions about the input tables and is always applicable. However, generic joins can degenerate to cross joins, which produce very large output results that do not fit in GPU memory. Since this join type is typically not used in CoGaDB, we implemented only a generic CPU hash join.

The most common join type in OLAP workloads is the PK-FK join. Since we know that for PK-FK joins the number of result rows is the number of foreign keys in the foreign key table, we use an optimized version of the indexed-nested-loop join of He and others [17] for the GPU. The algorithm first sorts the primary-key column and, second, assigns all threads a number of foreign keys, which are looked up in the sorted primary key column using binary search. Therefore, we skip the phase where each thread first counts their matching result tuples. On the CPU, we use a hash join, where we build the hash table serially, and perform the pruning phase in parallel.

For OLAP queries, it is often more efficient to pre-filter a dimension table, before performing a join with the fact table. However, in this case, CoGaDB cannot use a PK-FK join, because the PK-FK relationship may be broken. However, we can pre-compute a join index, and use the matching TIDs of the dimension table to extract the matching TIDs from the fact table. This optimization proved to be very efficient, on the CPU and the GPU. Since a fetch join is basically a modified version of the merge step from a sort-merge join, we adapted the merging algorithm of He and others [17].

In case more than one join is involved in a query, CoGaDB checks whether the join can be combined in a star join [31], where n dimension tables are joined with a fact table. We provide more details on CoGaDB's star join in Sect. 4.4.

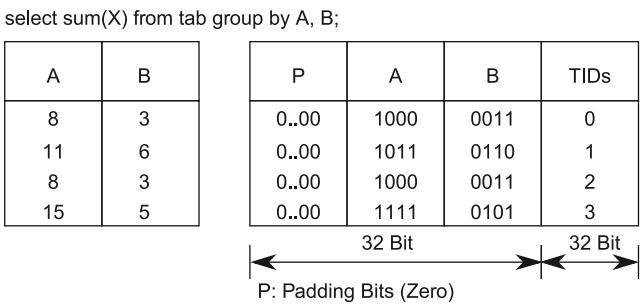

Fig. 3 Packing values of multiple columns to group a table by columns A and B with a single sorting step

4.1.4 Sorting

Sorting is an important building block of many operators, such as joins. CoGaDB uses the parallel sort primitive of Intel's TBB library on the CPU and the Thrust library on the GPU. For order by statements, CoGaDB needs to sort a table by multiple columns. For this, we start sorting a group of columns $A_1, \ldots, A_n$ first by A_n, then we retrieve the resulting TID list and fetch the values A_{n-1} to obtain the reordered version of A'_{n-1}. We continue this until we sorted A_1, which results in the final TID list that is the correct sorting order of all groups. Note that this requires a stable sorting algorithm. This approach may seem inefficient, but note that this primitive is used for the final table sort specified in SQL's order-by clause, which typically does not exceed several hundred tuples. Furthermore, we can perform the sorting on the CPU and the GPU.

4.1.5 Group By

CoGaDB uses sorting based grouping algorithms, one generic and one specialized algorithm. The generic algorithm uses the multi-column sort algorithm. A typical optimization is to pack a group of columns in a single 32-Bit integer (or another *bank size*) [22, 33]. Since we sort by the group keys and need the corresponding TID list, we need another 32 Bit as payload. Then, a 32-Bit group key is stored in the upper 32 Bit of a 64-Bit integer and the payload is stored in the lower 32 Bit. We illustrate this principle in Fig. 3. Then, we sort the array of 64-Bit values, extract the lower 32 Bit as result TID list, and obtain a correct grouping with a single sort operation, either on the CPU or the GPU.

4.1.6 Aggregation and Arithmetic Operations

Based on an input grouping, an aggregation combines data from a column using an aggregation function. CoGaDB supports as aggregation function SUM, COUNT, MIN, and MAX. For the CPU, we use a serial reduction function, whereas we use a parallel reduction function from the Thrust library on the GPU. Since many queries do not just apply an aggregation function, but also perform column algebra operations, we need to efficiently express complex aggregation functions (e.g., select sum(A+B/C) ...). Similar to other column stores (e.g., MonetDB [21]), CoGaDB has separate operators for column arithmetic and constructs for an algebra expression an operator tree that computes the result.

4.2 Materialization Strategy

In every column store, there is a time where the internal columnar representation has to be transformed to a row-wise representation. There are two basic options: Reconstructing tuples as early as possible (early materialization) or as late as possible (late materialization). Manegold and others [26] and Abadi and others [1] found that late materialization is more efficient than early materialization in case queries contain highly selective predicates, and the data is aggregated by a query [1], which is typically the case for OLAP queries. Thus, CoGaDB uses late materialization by transforming the result table in a row-oriented table.

4.3 Operator Parallelism

In a bulk processor, each operator can use intra-operator parallelism to increase its efficiency. Another form of parallelism is building bushy query execution plans, where independent sub-plans can be processed in parallel. CoGaDB exploits both types of parallelism, but, in our implementation, we gave priority to inter-operator parallelism, because high parallelism inside operators can lead to over-utilization of processors in case we have large bushy query plans. We expect that advanced approaches such as morsel-driven parallelism [23] or the admission control mechanism of DB2 BLU [34] can avoid over-utilization more efficiently.

While parallel execution of threads is a zero-effort solution on CPUs, we have to add another mechanism on GPUs. CUDA uses the concept of streams to structure parallelism between kernels. Two kernels can be executed in parallel, if and only if they are queued in two different CUDA streams and no kernel is assigned to the default stream 0 [30]. The same principle goes for interleaving copy operations with kernel executions, another important optimization for GPU-accelerated DBMSs.

We achieve inter-operator parallelism on GPUs by creating n CUDA streams managed by a *stream manager*. Each GPU operator requests a stream, and the stream manager assigns a stream from a fixed set of streams using the round-robin strategy.

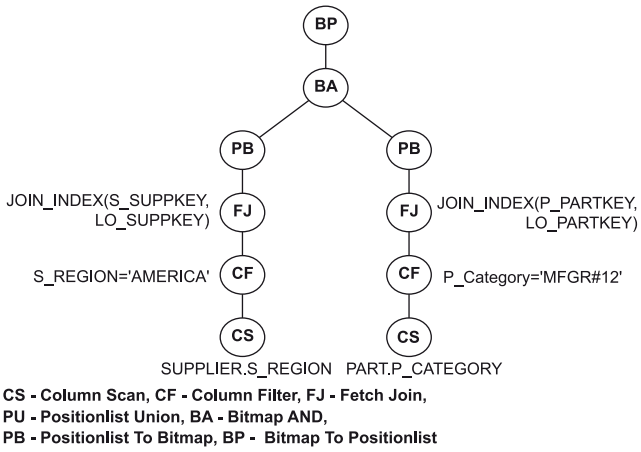

Fig. 4 Query plan for star join for SSBM query 2.1

4.4 Parallel Star Joins

In case a query contains multiple joins between the fact table and the dimension tables, the joins can be rewritten into a star join, which can process each join between the fact table and a dimension table in parallel [31]. Abadi and others proposed the invisible join [2], an extension of O'Neil's approach [31]. The invisible join is a late materialized join, which reduces the number of expensive random accesses on dimension tables (e.g., with an unsorted position list) [1].

The invisible join works in three phases. First, the predicates of the query are applied to the dimension tables. For each dimension table, we get a list of dimension table keys and insert them into a hash table. In the second step, the corresponding foreign key in the fact table are pruned in the hash table of the corresponding dimension and produces a bitmap indicating the matching rows of the fact table for one dimension. Then, the bitmaps of all dimensions are combined using a bitwise AND operation. In the third phase, the matching rows in the fact tables are looked up in the dimension tables to construct the result of the star join.

CoGaDB implements a variant of the invisible join, where the building of the hash table and pruning of the fact table is replaced by a fetch join from a join index. This significantly improved the performance, especially for large dimension tables. The extracted keys from the join index are converted in a bitmap and combined by a bitwise AND operation. The resulting bitmap is converted back to a position list, which is the output format expected by CoGaDB's query processor. In a final step, the result table of the star join is computed by joining the filtered fact table with the filtered dimension tables. CoGaDB's query processor can perform any step from phase one and two of the invisible join on the CPU or the GPU. We illustrate the first two steps in Fig. 4.

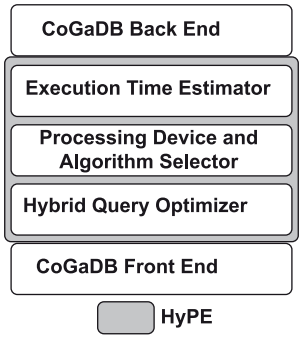

Fig. 5 Architecture of HyPE, taken from [11]

Since query plans for the star join operators are very large, bushy trees, CoGaDB's query optimizer executes different parts of the invisible join on the CPU and the GPU, which leads to inter-device parallelism.

5 Hybrid Query Optimizer

Up to now, we discussed how CoGaDB's query processor executes queries. In a heterogeneous processor system, it is crucial that the processed plan makes efficient use of the computational resources and available memory bandwidth on all processors. In order to achieve this goal, CoGaDB uses our *Hybrid Query Processing Engine* (HyPE) [9, 11] for the physical optimization and operator placement. HyPE consists of three components: the estimation component, the algorithm selector, and the hybrid query optimizer (cf. Fig. 5).

5.1 Estimation Component

HyPE aims to be hardware- and algorithm-oblivious, which means that it requires minimal knowledge of the underlying processors or the implementation details of database operators. To achieve this goal, the estimation component uses simple regression models (e.g., least squares) to approximate the performance behavior of database algorithms on different hardware w.r.t. properties of the input data (e.g., data size or selectivity). The algorithm runtimes of the first queries executed by CoGaDB serve as training data, and HyPE continuously monitors algorithm runtimes of queries and refines cost models at runtime to improve the model's accuracy. Since CoGaDB uses a bulk processor, the overhead of this continuous monitoring and adaption is minimal, because it is done once for each operator invocation.

5.2 Processor Allocation and Algorithm Selection

Based on the cost estimator, the optimizer needs to decide for each operator in a query plan, on which processor it should execute the operator, and which algorithm should be used [10]. For each operation, the available algorithms are fetched from a pool of algorithms. Then, an estimation component computes for each algorithm an estimated execution time. Finally, a decision component selects an algorithm according to a user-specified optimization heuristic. After the algorithm finished execution, it returns its execution time to the estimation component in order to refine future estimations. We summarize HyPE's decision model in Fig. 6. Thus, HyPE solves the processor allocation and the algorithm-selection problem in a single step, because it decides for a certain algorithm, which is specific to a certain processor type (e.g., CPU or GPU). In case of multiple devices, HyPE assigns unique identifiers to each algorithm that allows us to pin-point which processor belongs to which algorithm. The optimal algorithm is selected according to an optimization strategy. By default, HyPE uses *Waiting Time Aware Response Time* (WTAR) [12], a scheduling strategy that considers the load on each processor and the estimated execution times of all algorithms for a certain operator. HyPE keeps track, on which operator was assigned to which processor, and uses the accumulated estimated execution times of all operators inside a ready queue as measure for the load condition on a processor, as illustrated by Fig. 7. The processor that is expected to have the minimal response time is used to execute the operator [12]. This allows us to automatically balance the operators in a query plan on available processors.

5.3 Hybrid Query Optimization

The query optimizer assigns for each operator in a query plan a suitable target processor and algorithm. HyPE supports two query optimization modes. In the first mode, HyPE traverses the query plan and requests operator placements from the algorithm selector [9]. Although this is a greedy strategy, it does consider the load on the processors, in case the WTAR optimization strategy is used. Interestingly, the greedy strategy coupled with WTAR schedules queries in a way that independent sub-plans are evaluated in parallel on different processors, which can lead to significant performance gains. In the second mode, HyPE creates a set of candidate plans and performs a classical cost-based optimization, where the costs are not cardinalities, but (estimated) execution times. This approach often suffers from poor cardinality estimates, but this problem is not specific to CoGaDB, it is inherent in all DBMSs.

6 Performance Evaluation

After describing the architectural design considerations and implementation details, we now conduct a performance evaluation in order to show the efficiency of CoGaDB.

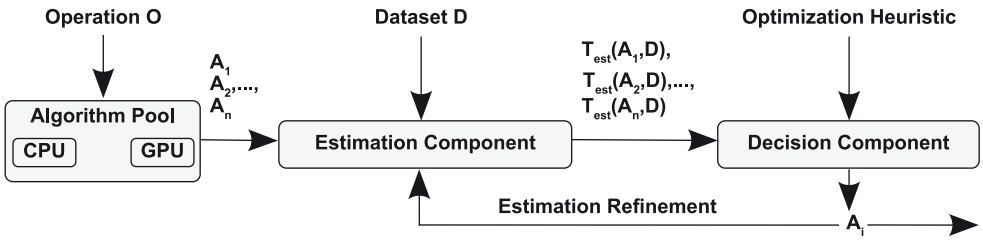

Fig. 6 HyPE's underlying decision model, taken from [10]

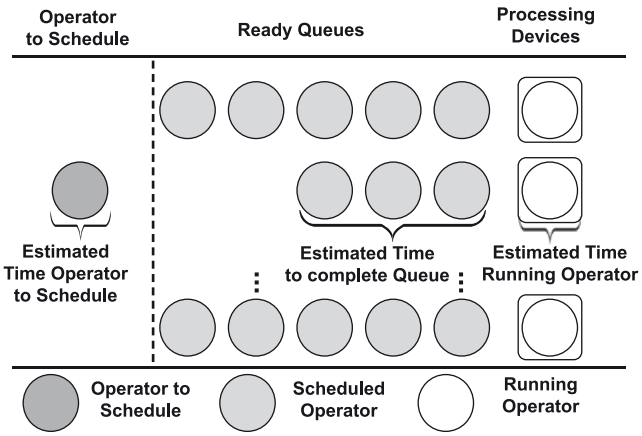

Fig. 7 Load Tracking with Ready Queues, taken from [12]

6.1 Evaluation Setup

We conduct our experiments on a machine with an Intel® Core™i7-4770 CPU having 4 cores (@3.40 GHz) with 24-GB of main memory and a GeForce® GTX 660 GPU (@980 MHz) with 1.5-GB device memory.[6] Since CoGaDB is optimized for OLAP benchmarks, we use the Star Schema Benchmark [32], which is a popular OLAP benchmark frequently used for performance evaluations [23, 38]. For all experiments, we use one database with scale factor 15. Therefore, the complete database does not fit into the GPUs memory.

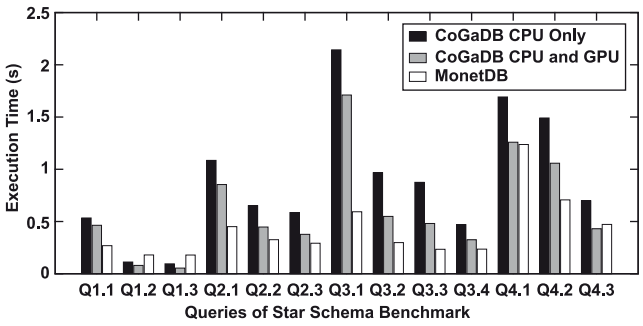

Fig. 8 Response times of CoGaDB for the SSBM with scale factor 15

[6]Hyper-threading was enabled during our experiments.

6.2 Experiments

In our experiments, we want to answer two questions:

1. Can GPU acceleration significantly improve the performance of query processing, even if the database does not fit in the GPU memory?
2. Can we achieve stable performance in a GPU-accelerated DBMS, in case we use a learning-based optimizer with no detailed information about the processors of a machine?

6.2.1 Performance Gain of GPU Acceleration

We executed a workload containing all SSBM queries 100 times and executed first all queries of the benchmark. Then, we repeated this process until we executed all queries 100 times. Note that the first run of the SSBM workload was used as warm-up queries, where performance is not considered. We computed the average of the remaining 99 measurements and show the performance of CoGaDB for each query of the SSBM in Fig. 8 in CPU-only mode and with GPU acceleration. As baseline, we included the results of MonetDB, a highly optimized main-memory DBMS [21].

We observe that CoGaDB's performance is significantly improved by GPU acceleration, where query 3.3 benefits the most from the GPU (by factor 1.8), whereas query 1.1 has the least benefit from GPU acceleration (by factor 1.15). Furthermore, we compare the performance of CoGaDB with MonetDB. For the experiments with MonetDB, we used MonetDB 11.17.13 and optimized it for performance as follows. First, we configured MonetDB to be compiled with optimization and without debugging. Second, we set the database to read-only mode. This allows MonetDB to use more efficient MAL plans. Finally, we set the OID size to 32 Bit to be comparable with the 32-Bit TID size of CoGaDB.

We can see that CoGaDB's performance is in the same order of magnitude as that of MonetDB. Hence, CoGaDB is a suitable evaluation platform for query optimization and load balancing in heterogeneous processor systems. However, MonetDB is still significantly faster for some queries (e.g., Q1.1, Q2.1, Q3.1-Q3.3, and Q4.2) even if we enable GPU acceleration in CoGaDB. This is because MonetDB is a highly optimized system that contains many heavily tuned

algorithms. If these optimizations would be included in CoGaDB, we could expect CoGaDB to be as fast as (or even outperform) MonetDB. There are two more major reasons for the difference in performance. The first is in the way the two engines parallelize queries, and the second is the data transfer bottleneck.

Parallelization Strategies. MonetDB uses a technique called mitosis, where the query plan is replicated and each thread processes its own plan, which is pinned to a fraction (horizontal partition) of the input BATs. CoGaDB evaluates each child in parallel and, therefore, the performance is bound by the longest path in the query plan. Hence, during query execution, CoGaDB does typically not (yet) utilize all cores to 100 %, because not all CPU operators have a parallel version (yet). We discuss possible extensions to improve CoGaDB's performance in Sect. 7.

Data Transfer Bottleneck. CoGaDB is not generally faster than MonetDB, even though it uses an additional processor with significantly higher raw processing power than a CPU. Compared to MonetDB, which uses the CPU only, this seems like a poor result. However, the performance of query processing on GPUs does not simply scale with the number of cores of a GPU. The main problem is the data transfer between the CPU and the GPU [16].

Since CoGaDB makes heavy use of join indexes (similar to MonetDB or MonetDB's OpenCL Extension Ocelot [19]), it also suffers a performance penalty in case we need to copy the index first (but if it is cached in the GPUs memory, the speedup is significant).

On our test machine for the used scale factor of 15, a join index needs a memory capacity of 686 MB. Since CoGaDB uses half of the GPU's memory for buffering ($\approx$ 750 MB) to leave enough free memory for temporary data structures and results of GPU operators, only one join index fits in the GPU buffer at a time, which limits the benefit of the GPU: With more device memory, the performance of CoGaDB would increase as well. It is part of our ongoing research to reduce this negative impact of the data transfer bottleneck.

The data transfer bottleneck is bidirectional, so CoGaDB's performance suffers also from large intermediate results, which need to be transferred back to the CPU. For all queries were CoGaDB performs poorly compared to MonetDB, the query selectivity is relatively small. For the invisible join, this means that larger position lists have to be intersected, which leads to higher intersection costs and depending on the query plan, higher data transfer costs.

6.2.2 *Benefit of Adaptive Physical Query Optimizer*

We now conduct experiments to answer research question 2. We executed the same workload from the previous experiment, but, this time, we visualize the query execution times of queries Q1.1, Q2.1, Q3.1, and Q4.1 over time in Fig. 9. For the first ten executions, the execution times of queries have a high variance. After this initial phase, the execution times remain stable. During the unstable phase, CoGaDB has no cost models for all processors, and assigns processors to operators using a round-robin strategy. After performance models become available, CoGaDB chooses the processor (and algorithm) that are optimal according to the learned cost models. Therefore, the performance gradually improves over time and remains stable.

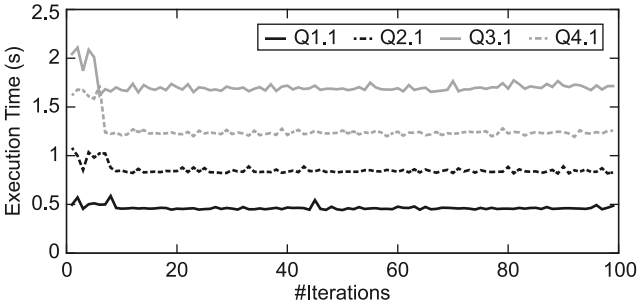

Fig. 9 Response times of selected SSBM queries in CoGaDB over 100 executions

7 Future Development

In this section, we describe future developments on CoGaDB: efficient algorithms, support of the CUBE operator and other co-processors, and alternative query processing and cardinality estimation approaches.

Efficient Algorithms Although we invested much time in tuning CoGaDB's database algorithms on the CPU and the GPU, the primary focus was still in exploiting the heterogeneous nature of the modern hardware landscape and, thus, on cost estimation and load-aware processor allocation. However, for future work, we will adapt approaches for efficient joins [5, 6] and aggregations [37]. Here, we have two implementation choices: Using hardware-oblivious operators written in a processor-independent language (e.g., OpenCL [19]) or tailoring algorithms for every processor type [13, 14].

CUBE Operator The CUBE operator [15] is a compute-intensive operator, which is frequently used in OLAP scenarios. Hence, it would be beneficial to investigate the potential performance gains by offloading parts of the computation to GPUs.

Support for Other Co-Processors Aside GPUs, other architectures have merged for co-processors such as *Multiple Integrated Cores* MICs (e.g., Intel Xeon Phi). It would be interesting to investigate the performance pro-

perties of MICs for DBMSs to identify the optimal (co-) processor for a certain task or workload.

Query Processing Strategies Aside from tuple-at-a-time volcano-style and operator-at-a-time bulk processing, there are alternative query processing strategies such as query compilation [29] or vectorized execution [7]. It is not yet clear which strategy is optimal for heterogeneous processor environments. For a fair comparison, all strategies should be implemented in a single system.

Cardinality Estimation Our query optimizer relies on accurate cardinality estimates, which still poses major problems. Markl and others developed progressive optimization, a technique where checkpoints are inserted in the query plan [27]. In case the cardinality estimates are too inaccurate at a checkpoint, a re-optimization is triggered. Stillger and others proposed LEO, DB2's learning optimizer, which continuously monitors cardinality estimations and iteratively corrects statistics and cardinality estimations [35]. Heimel and Markl offloaded selectivity estimation to the GPU, which allows them to use more compute-intensive approaches to increase estimation accuracy [18].

8 Related Work and Systems

In this section, we will discuss related systems. Yuan and others study the performance behavior of OLAP queries on GPUs with their system GPUDB [38] that compiles queries to driver programs, which call pre-implemented GPU operators. Thus, GPUDB performs only dispatcher and post-processing tasks on the CPU. Wang and others developed MultiQx-GPU [36], an extension of GPUDB that can process several queries in parallel on GPUs.

He and others develop GPUQP, the first DBMS accelerated by GPUs [17]. Similar to CoGaDB, GPUQP can execute each operator on the CPU or the GPU. However, GPUQP uses analytical cost models to decide on an operator placement, whereas CoGaDB relies on learned cost models and runtime refinement. Based on GPUQP, Zhang and others develop OmniDB [39], where the main focus is to exploit all heterogeneous processors while keeping a maintainable code base. They use an architecture based on adapters to decouple the database kernel from the operators.

The same goal is addressed by Ocelot [19], a hardware-oblivious database engine that extends MonetDB by a set of OpenCL operators, which can be dynamically compiled to any OpenCL-compliant device, such as CPUs, GPUs, or Xeon Phis. Thus, Ocelot leaves the handling of heterogeneity to the hardware vendors, while maintaining high performance.

Bakkum and Chakradhar developed Virginian, whose main focus is high efficiency of database queries on GPUs [4]. Therefore, it implements the opcode model, where each operator is associated with a unique id and called from a management GPU kernel. This allows Virginian to execute any query with a single GPU kernel. Mühlbauer and others developed a heterogeneity-conscious job-to-core mapping approach [28], similar to the scheduler in HyPE [9, 12].

9 Conclusion

The power wall forces hardware vendors to specialize processors for certain tasks, which increases the heterogeneity of the hardware landscape. This, in turn, means that database vendors have to adjust their architectures to such heterogeneous environments.

In this paper, we presented CoGaDB, a system designed to target the hardware heterogeneity on the query optimizer level. We outlined design decisions and implementation details of CoGaDB and discussed how we combine a modern, GPU-accelerated DBMS with a hardware-oblivious query optimizer.

Our evaluation shows that the design, where an optimizer has no detailed knowledge of the hardware, is feasible. Furthermore, we showed that such a system can be competitive to highly optimized main-memory databases such as MonetDB.

Acknowledgments We thank Jens Teubner from TU Dortmund University and Theo Härder from University of Kaiserslautern for their helpful feedback.

References

1. Abadi D, Myers D, DeWitt D, Madden S. (2007) Materialization strategies in a column-oriented DBMS. In: ICDE, IEEE, pp 466–475
2. Abadi DJ, Madden SR, Hachem N. (2008) Column-stores vs. row-stores: how different are they really? In: SIGMOD, ACM, pp 967–980
3. Abadi D, Boncz P, Harizopoulos S, Idreos S, Madden S (2013) The design and implementation of modern column-oriented database systems. Foundations Trends in Databases 5(3):197–280
4. Bakkum P, Chakradhar S (2012) Efficient data management for GPU databases. http://pbbakkum.com/virginian/paper.pdf.
5. Balkesen C, Alonso G, Teubner J, Özsu MT (2013) Multi-core, main-memory joins: sort vs. hash revisited. PVLDB 7(1):85–96
6. Balkesen C, Teubner J, Alonso G, Özsu MT (2013) Main-memory hash joins on multi-core CPUs: tuning to the underlying hardware. In: ICDE, pp 362–373
7. Boncz PA, Zukowski M, Nes N (2005) MonetDB/X100: hyper-pipelining query execution. In: CIDR, pp 225–237
8. Borkar S, Chien AA (2011) The future of microprocessors. Commun ACM 54(5):67–77
9. Breß S, Geist I, Schallehn E, Mory M, Saake G (2012) A framework for cost based optimization of hybrid CPU/GPU query plans in database systems. Control Cybernetics 41(4):715–742

10. Breß S, Beier F, Rauhe H, Sattler K-U, Schallehn E, Saake G (2013) Efficient co-processor utilization in database query processing. Information Systems 38(8):1084–1096
11. Breß S, Heimel M, Saecker M, Köcher B, Markl V, Saake G (2014) Ocelot/HyPE: optimized data processing on heterogeneous hardware. PVLDB 7(13)
12. Breß S, Siegmund N, Heimel M, Saecker M, Lauer T, Bellatreche L, Saake G (2014) Load-aware inter-co-processor parallelism in database query processing. Data & Knowledge Engineering . doi:10.1016/j.datak.2014.07.003
13. Broneske D, Breß S, Heimel M, Saake G (2014) Toward hardware-sensitive database operations. In: EDBT, OpenProceedings.org, pp 229–234
14. Broneske D, Breß S, Saake G (2014) Database scan variants on modern CPUs: a performance study. In: IMDM@VLDB
15. Gray J et al. (1997) Data cube: a relational aggregation operator generalizing group-by, cross-tab, and sub-totals. Data Min Knowl Disc 1(1):29–53
16. Gregg C, Hazelwood K (2011) Where is the data? Why you cannot debate CPU vs. GPU performance without the answer. In: ISPASS, IEEE, pp 134–144
17. He B, Lu M, Yang K, Fang R, Govindaraju NK, Luo Q, Sander PV (2009) Relational query co-processing on graphics processors. ACM Trans Database Syst 34:21
18. Heimel M, Markl V (2012) A first step towards GPU-assisted query optimization. In: ADMS, pp 33–44
19. Heimel M, Saecker M, Pirk H, Manegold S, Markl V (2013) Hardware-oblivious parallelism for in-memory column-stores. PVLDB 6(9):709–720
20. Heimel M, Haase F, Meinke M, Breß S, Saecker M, Markl V (2014) Demonstrating self-learning algorithm adaptivity in a hardware-oblivious database engine. In: EDBT, OpenProceedings.org, pp 616–619
21. Idreos S, Groffen F, Nes N, Manegold S, Mullender KS, Kersten ML (2012) MonetDB: two decades of research in column-oriented database architectures. IEEE Data Eng Bull 35(1):40–45
22. Johnson R, Raman V, Sidle R, Swart G (2008) Row-wise parallel predicate evaluation. PVLDB 1(1):622–634
23. Leis V, Boncz P, Kemper A, Neumann T (2014) Morsel-driven parallelism: a NUMA-aware query evaluation framework for the many-core age. In: SIGMOD, ACM, pp 743–754
24. Manegold S, Boncz PA, Kersten ML (2000) Optimizing database architecture for the new bottleneck: memory access. VLDB J 9(3):231–246
25. Manegold S, Boncz P, Kersten ML (2002) Generic database cost models for hierarchical memory systems. In: PVLDB, VLDB Endowment, pp 191–202
26. Manegold S, Boncz P, Nes N, Kersten M (2004) Cache-conscious radix-decluster projections. VLDB, VLDB Endowment, pp 684–695
27. Markl V, Raman V, Simmen D, Lohman G, Pirahesh H, Cilimdzic M (2004) Robust query processing through progressive optimization. In: SIGMOD, ACM, pp 659–670
28. Mühlbauer T, Rödiger W, Seilbeck R, Kemper A, Neumann T (2014) Heterogeneity-conscious parallel query execution: getting a better mileage while driving faster! In: DaMoN, ACM, pp 2:1–2:10
29. Neumann T (2011) Efficiently compiling efficient query plans for modern hardware. PVLDB 4(9):539–550
30. NVIDIA. NVIDIA CUDA C Programming Guide. (2014) http://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf. pp 31–36, Version 6.0. Accessed 18 May 2014
31. O'Neil P, Graefe G (1995) Multi-table joins through bitmapped join indices. SIGMOD Rec 24(3):8–11
32. O'Neil P, O'Neil EJ, Chen X (2009) The star schema benchmark (SSB), Revision 3. http://www.cs.umb.edu/poneil/StarSchemaB.PDF
33. Raman V, Swart G, Qiao L, Reiss F, Dialani V, Kossmann D, Narang I, Sidle R (2008) Constant-time query processing. In: ICDE, IEEE, pp 60–69
34. Raman V et al (2013) DB2 with BLU acceleration: so much more than just a column store. PVLDB 6(11):1080–1091
35. Stillger M, Lohman GM, Markl V, Kandil M (2001) LEO - DB2's learning optimizer. In: VLDB, Morgan Kaufmann Publishers Inc., pp 19–28
36. Wang K, Zhang K, Yuan Y, Ma S, Lee R, Ding X, Zhang X (2014) Concurrent analytical query processing with GPUs. PVLDB 7(11):1011–1022
37. Ye Y, Ross KA, Vesdapunt N (2011) Scalable aggregation on multicore processors. In: DaMoN, ACM, pp 1–9
38. Yuan Y, Lee R, Zhang X (2013) The yin and yang of processing data warehousing queries on GPU devices. PVLDB 6(10):817–828
39. Zhang S, He J, He B, Lu M (2013) OmniDB: towards portable and efficient query processing on parallel CPU/GPU architectures. PVLDB 6(12):1374–1377
40. Zhou J, Ross KA (2002) Implementing database operations using SIMD instructions. In: SIGMOD, ACM, pp 145–156