Tomas Karnagel, Dirk Habich, Benjamin Schlegel, Wolfgang Lehner

Heterogeneity-Aware Operator Placement in Column-Store DBMS

Erstveröffentlichung in / First published in:

Datenbank-Spektrum. 2014, 14 (3), S. 211–221 [Zugriff am: 02.02.2023]. Springer. ISSN 1610-1995.

DOI: http://dx.doi.org/10.1007/s13222-014-0167-9

Final edited form was published in "Datenbank-Spektrum." 14 (3), S. 211–221 . ISSN: 1610-1995
http://dx.doi.org/10.1007/s13222-014-0167-9

Heterogeneity-Aware Operator Placement in Column-Store DBMS

Tomas Karnagel · Dirk Habich · Benjamin Schlegel · Wolfgang Lehner

Abstract Due to the tremendous increase in the amount of data efficiently managed by current database systems, optimization is still one of the most challenging issues in database research. Today's query optimizer determine the most efficient composition of physical operators to execute a given SQL query, whereas the underlying hardware consists of a multi-core CPU. However, hardware systems are more and more shifting towards heterogeneity, combining a multi-core CPU with various computing units, e.g., GPU or FPGA cores. In order to efficiently utilize the provided performance capability of such heterogeneous hardware, the assignment of physical operators to computing units gains importance. In this paper, we propose a heterogeneity-aware physical operator placement strategy (HOP) for in-memory columnar database systems in a heterogeneous environment. Our placement approach takes operators from the physical query execution plan as an input and assigns them to computing units using a cost model at runtime. To enable this runtime decision, our cost model uses the characteristics of the computing units, execution properties of the operators, as well as runtime data to estimate execution costs for each unit. We evaluated our approach on full TPC-H queries within a prototype database engine. As we are going to show, the placement in a heterogeneous hardware system has a high influence on query performance.

T. Karnagel (✉) · D. Habich · B. Schlegel · W. Lehner
Technische Universität Dresden,
Nöthnitzer Str. 46,
01187 Dresden, Germany
e-mail: tomas.karnagel@tu-dresden.de

D. Habich
e-mail: dirk.habich@tu-dresden.de

B. Schlegel
e-mail: benjamin.schlegel@tu-dresden.de

W. Lehner
e-mail: wolfgang.lehner@tu-dresden.de

1 Introduction

For the last 30 years, disk-centric database systems based on commodity hardware have reflected the state of the art. Within the last few years, however, this architecture has dramatically changed due to several reasons, but especially due to significant developments in the hardware sector. Because of the general availability of large main-memory capacities, columnar in-memory database systems become more and more popular since the entire data fits into main memory. In this approach, the key characteristics of low access latency and high bandwidth of main memory can be efficiently exploited for complex analytic query processing.

Besides the availability of high main-memory capacities and multi-core CPUs, hardware systems are more and more shifting towards heterogeneity. That means a multi-core CPU with large main memory is packed into one single hardware box together with one or more additional computing units, e.g., GPU or FPGA cores. Fundamentally, co-processor architectures have already combined a common CPU with an accelerator like GPU or FPGA, so that heterogeneity is nothing new. However, these co-processor architectures are rather loosely-coupled due to the fact that each computing unit has its own isolated memory block and data has to be explicitly transferred between the computing units. A new trend in this hardware domain is the tightly-coupled combination of a common multi-core CPU with different computing units with a large unified memory hierarchy, where CPU and each accelerator have direct access and no explicit data transfer is necessary.

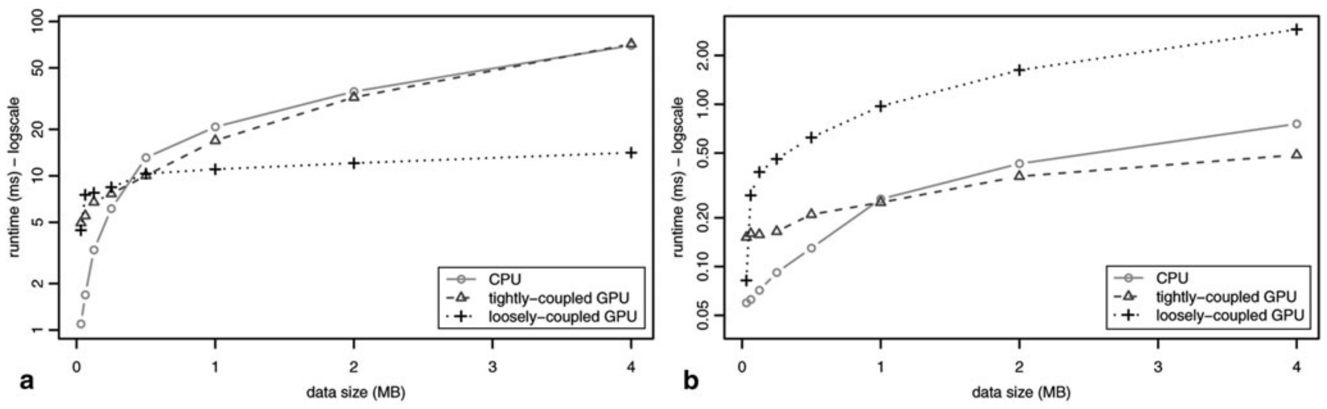

Fig. 1 Showing the need for heterogeneous placement decisions for two database operators. **a** Sorting Operator for different data sizes. **b** Selection Operator for different data sizes

In order to efficiently utilize the provided computing power of heterogeneous hardware, the major challenge for the data-base community is to design a heterogeneity-aware database system. A significant number of research work has already transformed single traditional database operators to loosely-coupled computing units [5, 6]. However, to tackle the heterogeneity aspect in the query processing context more generally, a generic physical operator approach is necessary. A first step in this direction has been done by Heimel et al. [8]. The authors propose a hardware-oblivious physical operator approach for an in-memory columnar data-base system, where they move from specific carefully optimized accelerator code to portable operator code, which is executable on different computing units. As they have shown, this approach exploited the full potential of modern parallel hardware architectures like multi-core CPUs and highly parallel GPUs.

However, a hardware-oblivious database system is not aware of the heterogeneous hardware environment. We believe that query processing has to be extended by assigning physical operators to appropriate computing units with regard to performance. To illustrate this aspect, we considered two simple database operators: sorting and selection. We executed both operators in columnar fashion on different computing units with all data in main memory: CPU, tightly-coupled GPU, and loosely-coupled GPU with explicit data transfer. In this experiment, we vary the size of the corresponding column, influencing data size and parallelism in execution. Figure 1(a) shows the runtime results for the sorting operator. For smaller data sizes, the CPU performs much better than both GPUs. With an increasing number of parallelism and data, the loosely-coupled GPU is more appropriate with regard to runtime. In this case, the additional data transfer between CPU and GPU is compensated by the full utilization of the GPU's high computing power. Figure 1(b) depicts the runtime results for the selection operator. Again, the CPU is more suited for the smaller data sizes, while a GPU is more appropriate for larger ones. In this case, however, the tightly-coupled GPU should be chosen over the loosely-coupled GPU. This is caused by the small overall runtime of this operator, where data transfers to the loosely-coupled GPU can not be compensated and yield a higher impact than the actual computation. In both test scenarios, the best computing unit does not change for larger data sizes beyond 2 MB. However, this is highly dependent on the operators and the hardware setup.

These two simple experiments indicate the benefits and challenges for query processing on heterogeneous hardware. Based on different characteristics like parallelism, data size, computational complexity of operators, and computing units properties, the execution times of physical operators on different computing units differ significantly. To tackle this aspect more precisely for query processing in columnar database systems, we propose our *HOP* approach for Heterogeneity-aware physical Operator Placement. *HOP* takes a physical query execution plan determined by a query optimizer and assigns the physical operators to available computing units using a cost model at runtime. In detail, our key contributions are:

- We present the necessity and the potential of heterogeneous computing for query processing together with a possible architecture of a database system supporting heterogeneous placement decisions.
- We propose a cost-model to estimate execution costs based on hardware characteristics and runtime information, as well as the operator's execution behavior.
- Finally, we present challenges beyond heterogeneous placement like parallelism between computing units and the support for multiple optimized operators per computing unit, together with discussing ideas for future work.

Final edited form was published in "Datenbank-Spektrum." 14 (3), S. 211–221 . ISSN: 1610-1995
http://dx.doi.org/10.1007/s13222-014-0167-9

Please note, that the actual integration into a given database system like Ocelot [8] is not the focus of this paper. First, we have to evaluate possible placement and integration techniques before actual integration can take place.

2 System Architecture

A significant number of research activities has already ported some traditional database operators to accelerators like GPU [6, 5] or FPGA [11]. However, it is required that research activities investigate more complete system support. From our point of view, a generalized placement decision for all query operators to all available computing units is a key challenge for database systems on heterogeneous hardware. The simple experiment in the introduction has shown that the placement has a high influence on execution performance and that it depends on various influencing factors. Therefore, we propose our HOP approach to assign arbitrary physical query operators to different computing units. Before we are going to describe our HOP approach in detail, we have to define our target database system as well as our chosen placement strategy.

Database System

To narrow research in this novel database field, we define our database system first. It is possible to weaken some of our assumptions for further optimization, but we focus on them as a starting point.

1. Our main focus is on in-memory columnar database systems [12, 3]. Moreover, physical operators are executed in a one-column-at-the-time approach.
2. The various computing units are used only for computation. We do not utilize computing units as an extended data store, leaving main memory as the central point where all data is stored. That means, for each operator, data have to be accessed or copied from main memory and the results have to be stored in main memory.
3. Each operator is executable on each computing unit of the heterogeneous system.

We limit our work in Point 1, because row-based execution would involve larger tuples and more working data, influencing transfers and data access. Also pipe-lined approaches would result in many small operator executions showing a significant overhead when using accelerators. We defined Point 2 for two reasons: with no data cached on the accelerators, queries can work on the most recent data from main memory and the accelerator caches are free to store only the operational data, being able to work on larger data sets. Point 3 is a basic requirement to allow heterogeneous placement.

Placement Optimization Strategy

Based on our defined database system, we are now able to specify our placement approach in more detail. Starting by an SQL query, the regular query optimizer attempts to determine the most efficient way to execute the query at compile time. Today, this query execution plan (QEP) is executed to compute the query result. Using our columnar database system, we are able to introduce a new layer between query optimizer and query executor, which is responsible for the operator placement in a heterogeneous hardware environment. Our novel placement layer takes the determined QEP as input and outputs the QEP with an assigned computing unit for each physical operator. In this case, we assume that the most efficient QEP is independent from heterogeneous hardware and the placement can be considered separately. In general, two aspects have a high influence on the complexity of the placement layer:

Placement Time: The placement decision can be done at compile time or runtime. While at compile time different characteristics like input data sizes of operators have to be estimated, at runtime this information can be precisely monitored. From an accurate point view, the runtime placement should therefore be preferred.

Placement Object: The placement can be either done individually for each physical operator or for complete sub-graphs of the QEP. The latter could be beneficial to support explicit parallelism between computing units or data placements. However, this would introduce dependencies between the placement decisions, which leads to an exploding search space. For example, a QEP with 10 operators that have to be placed on 3 computing units, results in $3^{10} = 59,049$ possible placements, which have to be evaluated. Without taking dependency restrictions into account, the search space is limited to number of available computing units for each operator execution.

In order to determine an accurate placement decision, our HOP approach determines the placement for each physical operator *individually at runtime*. From our point of view, this runtime approach is the most suitable approach for our columnar database system.

3 HOP Model

In order to conduct a runtime placement decision for each arbitrary physical operator individually, our HOP model consists of (i) an operator execution model to describe arbitrary physical database operators, whereas we concentrated on the time-consuming parts of the operator executions, and (ii) a cost function to estimate the operator's runtime on various

Final edited form was published in "Datenbank-Spektrum." 14 (3), S. 211–221 . ISSN: 1610-1995
http://dx.doi.org/10.1007/s13222-014-0167-9

computing units. Using our operator execution model and cost function, we are able to select the computing unit with the least estimated runtime for each physical operator. In the following, we describe our operator model as well as our developed cost function in detail.

Operator Execution Model
To describe the execution of arbitrary physical operators on various computing units and to estimate the operator runtimes on different computing units three different aspects play an important role.

Latency: Each physical operator execution on any computing unit has a certain amount of latency overhead. Generally, this latency overhead is independent from the physical operator and depends only on the computing unit.

Data Transfer: To execute an operator on a computing unit, the operator requires access to the appropriate data. If the computing unit cannot access main memory directly, input and output data have to be transferred to and from the unit. This can have an especially large impact on the overall runtime of the operator, whereas the impact depends on the connection type and the input and output data sizes.

Execution: Aside from latency and data transfer, the operator needs to be executed on the computing unit. To characterize the execution and to estimate the runtime, we require historical data of previous executions or general heuristics.

The most challenging issue is to characterize the execution part of operators. To tackle that issue, we propose an online learning approach (similar to Breß et al. [4]) combined with additional rules and heuristics for decisions without previous knowledge of executions on every execution unit. Using the heuristics, we do not require a calibration step for the operator execution.

Cost Estimation
Based on the previously introduced operator model, we adapt the cost function proposed by He et al. [6] by adding the latency time.

$$\begin{aligned}\textit{exec. estimate} &= \textit{latency} + \textit{transfer input}\\ &\quad + \textit{computation} + \textit{transfer output}\end{aligned}$$

Additionally to adding latency, our approach is defined by a different way to estimate the computation as well as a more detailed transfer estimate, where the bandwidth is not assumed to be constant.

Latency and Transfer
For the latency overhead, we assume a fairly constant amount of time, which is independent of the operator but dependent on the computing unit. The overhead can be measured beforehand. Data transfers to and from the computing unit can be modeled by:

$$\textit{transfer estimate} = \sum \frac{\textit{amount of data}}{\textit{transfer bandwidth for data}}$$

The amounts stand for input or output data on runtime and the bandwidth stands for the transfer bandwidth from host to the computing unit or from the computing unit to host. The transfer bandwidth is dependent on the amount of memory. Large data structures can be transferred efficiently in big blocks, while small structures experience low bandwidths. Since the transfer bandwidth is not constant, we need to calculate the transfer costs for each input and output memory object used in the operator, hence the sum in the equation. We can test the transfer bandwidth beforehand and approximate unknown memory sizes.

Execution Estimation
When subtracting the transfer and latency overheads from the runtime, the sole execution times on each computing unit show mainly a scaling behavior like linear scaling (e.g., selection, radix-sort) or exponential scaling (e.g., nested-loop join) according to the input data sizes and execution parallelism. Usually the possible parallelism in execution is dependent on the available input data for processing. In our approach, we use the amount of input data and the execution time, not including overheads, as historical data. When estimation is needed, we assume linear scaling between data points and return a runtime estimation from the weighted average of the two neighboring data points. To cope with no or small amounts of data points, we introduce three phases of estimation:

Phase 0 - Fig. 2a: No historical data is given and no estimation can be done.

Phase 1 - Fig. 2b: Only one data point is given. In this case, we use point (0,0) as second reference and assume linear behavior.

Phase 2 - Fig. 2c: More than one point is given. We do not use point (0,0) anymore and assume linear scaling between the points as well as before and after the known points.

In Phase 1, point (0,0) is essential since it is not possible to do any estimation from just one real data point. However, we do not use this point, if we have more information, because it influences the estimation for small data sizes too much. There, scaling might be largely influenced by not fully utilizing the computing unit, resulting in a scaling behavior similar to Fig. 2c. We choose a linear interpolation between the points to keep the model maintenance simple and the estimation reasonably fast. Exponential scaling will eventually be described when having enough data points. Another positive

Final edited form was published in "Datenbank-Spektrum." 14 (3), S. 211–221 . ISSN: 1610-1995
http://dx.doi.org/10.1007/s13222-014-0167-9

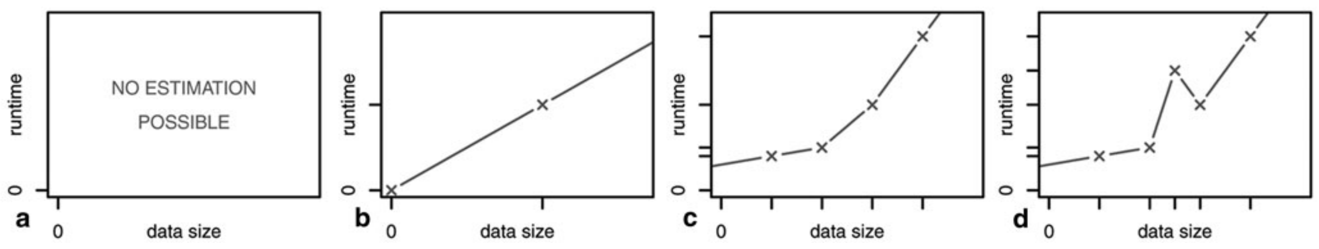

Fig. 2 Different phases of execution estimation depending on the known data. **(a)** Phase 0: no historical data **(b)** Phase 1: one data point **(c)** Phase 2: many data points **(d)** Phase 2 with outliers

effect on including only neighboring points into the estimation is the local impact of outliers (Fig. 2d). If we would use an approximation function for all data points (e.g., spline interpolation), outliers would have a global impact on the estimation instead of a local one in our case. However, it is advisable to do periodic cleaning of historical data to ensure that outlying data points are removed.

Placement Heuristics

The presented learning approach works well if we have at least one data point for each computing unit. However, this might not be the case in the beginning. Therefore, we developed heuristics to make a decision if execution estimation is not possible. For these heuristics we introduce two new properties of computing units, which can be measured beforehand. The first is the full utilization (FU) number, which is the amount of threads that fully utilizes the computing unit. For CPU systems, the processor is fully utilized when all hardware threads are busy. For GPU systems, the processors hide memory latency by fast context switching. In this case, the computing unit is fully utilized if enough threads are available to hide the memory latency. This leads to a high amount of threads for full utilization, even higher than the actual core count. We propose an easy way to estimate the amount of threads needed for full utilization in Sect. 4. The second property is the computational power (CP), which describes roughly the processing capabilities of the computing unit and can be measured beforehand. With these two properties, the knowledge about latency overheads (L) and transfer costs (T), as well as the given runtime configuration (thread count (th) and involved data (d)) and already given execution estimations (est), we can devise several rules:

1. We only consider computing units that are capable to execute the operator with the given runtime data (e.g., enough dedicated memory).
2. If the operator has never run before on any computing unit, we choose the unit that can be fully utilized ($th \geq FU$) with the lowest overheads ($min(L + T(d))$). If this leaves multiple computing units, we choose the most powerful one ($max(CP)$). If no computing unit can be fully utilized by th, we choose the computing unit closest to being fully utilized ($min(FU)$).
3. If we have a runtime estimation for one computing unit, we change to a different one, if the operator can fully utilize it ($th \geq FU$) and the estimated execution time for the first computing unit including overheads is higher than the new overheads ($(est_1 + L_1 + T_1(d)) > (L_2 + T_2(d))$).
4. If we have estimations for all computing units, we choose the computing unit with the smallest estimated overall runtime ($min(est + L + T(d))$).

The first and the last rule are trivial, while the second and third rule are heuristics for unknown execution estimations. The second rule might always pick the CPU or an iGPU[1] if available in the system. The decision between the two is mainly dependent on the latency and the full utilization point of the iGPU. The second heuristic makes sure that the computing unit is not changed, unless the estimated execution, in addition to any overheads, is higher than the overheads of a more powerful computing unit. There, we assume the execution to be faster than on the initial computing unit based on the computational power.

Related Work

There has been prior work in characterizing operator execution on different hardware platforms. For the CPU, cost models focus mainly on memory access [10] and can not be adapted to the GPU without adjustment. A GPU cost model was presented by He et al. [6] including transfer considerations, computation, and memory access estimation. The key difference to our approach is the execution estimation. The authors run their operators first as benchmarks to estimate their computation time and provide a memory access cost function for each operator primitive. Our HOP approach is meant to support arbitrary operators without defining the memory access patterns and it learns the execution time of operators during query processing. Additionally, we apply our model without much customization to the CPU or other com-

[1]An integrated GPU (iGPU) is tightly-coupled with a CPU and utilizes a portion of CPU-RAM rather than dedicated graphics memory.

puting units as well. He et al. also presented work on using a cost model for load balancing in tightly-coupled systems [7], where the focus is solely on hash joins. They used a static approach of instruction counts per tuple and instructions per cycle to estimate the computation. A more dynamic approach is presented by Breß et al. [4], where execution times are estimated using a learning approach with prior training and spline interpolation between the data points. However, the focus is on workloads that only change slowly and the system adapts to the workload over time. Despite many promising approaches in the past, our HOP approach and this paper defines itself by using a combination of static estimation for overheads, an online learning approach for the computation, and heuristics for unknown scenarios. Furthermore, it focuses on reasoning about integration techniques of our HOP approach for common database systems.

Outside the database research, multiple heterogeneous placement advisers are worth mentioning. In StarPU [2] for example, the authors use heterogeneous earliest-finish time scheduling to place tasks on computing units. Kicherer et al. [9] propose an online learning approach with a learning process at the beginning and a guided mode afterwards. When the estimations differ too much from the real values, the learning process starts again. The mentioned placement and scheduling approaches are application independent and do not consider database workloads or integration into a database system.

4 Obtaining HOP Parameters

In this section, we present ways to determine the computing unit properties to be used as parameters for our HOP model.

We measure the latency overhead empirically by executing an operator multiple times on the computing unit with synchronization between the operator calls compared to the same execution without synchronization. In the latter variant, all operations are scheduled together avoiding synchronization and latency overhead. The difference between the two variants show us the amount of latency overhead we have to expect on a single operation call for each computing unit. For GPUs, we have seen the latency being constant for multiple runs. For the CPU, the results range from negative to small positive numbers, indicating that the CPU has varying latency depending on other applications and system load. Therefore, we have defined the CPU latency as 0 ms.

The transfer bandwidth depends on the amount of data being transferred. We can approximate the transfer time by transferring different data sizes to the computing unit beforehand.

The full utilization point is gathered empirically only for GPUs. For the CPU, we use the number of supported hardware threads as the full utilization variable. For GPUs, this is not trivial, because much more threads are needed to hide memory latency. We devised a heavy computation benchmark with growing thread numbers. For small amounts of threads, the execution is not varying significantly. With thread numbers fully utilizing the computing unit and above, the execution takes noticeably longer. For our cost model, we define the point where the execution time scaling changes significantly as full utilization point and the number of threads needed to fully utilize the computing unit.

To determine the computational power, we simply run a heavy computational test case with high parallelism on all computing units. In detail, we do 100 modulo calculations per thread for 8.4 million threads. We use the inverse runtime as the computational power property.

5 Evaluation

For the evaluation of our HOP model, we use a heterogeneous hardware setup consisting of a multi-core CPU with a loosely-coupled and a tightly-coupled GPU. First, we use our test suite, as presented in Sect. 4, to characterize the given computing units. The measured properties of our heterogeneous hardware are shown together with general information in Table 1.

After the initial tests, we can apply our HOP model. In the first stages of using the model, the decisions rely heavily on the computing unit properties and the placement heuristics. Decisions can vary with the order of learned execution times, making it hard to evaluate. In a running database system, the model should be fully initialized for the majority of the time, meaning the model has learned at least two data points for each operator on each computing unit. Therefore, our evaluation is focusing on the initialized model. We apply our HOP model to operators of three TPC-H queries with different scale factors, to determine an optimal placement for each operator. We choose the TPC-H queries 3, 5, and 6 as a sample, since they use mostly standard operators and no extensive optimization is needed (as, for example, for sub-queries). We built a prototype execution engine using OpenCL where we can evaluate all three queries with different placement configurations.

Operator Implementation

We implemented multiple operators for our evaluation. The actual implementation is not the focus of our work, since the HOP model is able to make placement decisions for any kind of implementation. However, we present the design choices on the implementation for completeness. All operators are implemented in OpenCL and all implementations are optimized for high parallelism. However, the operators are not optimized towards a single hardware platform. The **selection operator** (σ) is operating on an input column,

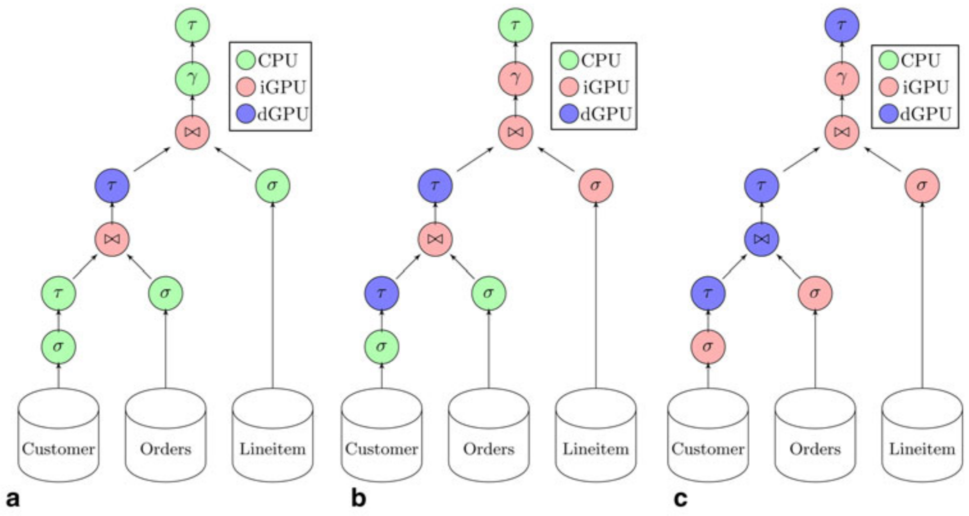

Fig. 3 Placement decisions for TPC-H query 3 with changing scale factor. **(a)** Scale factor 0.1 **(b)** Scale factor 1 **(c)** Scale factor 10

Table 1 Heterogeneous test system: AMD APU (CPU and iGPU) and Nvidia GPU as dedicated GPU (dGPU)

General information:	AMD CPU	A10-5800 K HD 7660D	NVIDIA K20
Computing Units[a]	4	6	13
Cores	4	384	2496
Frequency (MHz)	3800	800	706
Max R-BW (GB/s)	5.3	22.7	89.0
Max W-BW (GB/s)	7.4	18.0	143.8
Perf. (GFLOPS)	121.6	614.4	3524
Test suite results:			
Transfer BW (GB/s)	–	–	Max 3.1
FU (threads)	4	2048	9216
CP (1/s)	0.82	3.26	34.48
Latency (ms)	0	0.07	0.015

[a]Here, *Computing Unit* is used as OpenCL term, which usually means a clusters of cores

invalidating the entries not satisfying the conditions. The **sorting operator** (τ) is a parallel radix sort similar to the AMD SDK samples [1]. As **join operator** ($\bowtie$), we use a nested-loop join for small data sizes and an indexed-nested-loop join for larger data sizes. The index is created by sorting the smaller relation and using binary search for the index search. This is sufficient since all joins of our TPC-H test queries are based on primary-key/foreign-key referencing. The cardinality of the larger relation is also the upper bound of the join result. We choose the indexed-nested-loop join because any merging or building of complex hash tables is avoided, which is usually hard to parallelize efficiently. **Aggregation and grouping** (γ) is done through parallel reduction combined with hashing for group values.

Please note that we use one operator implementation for all computing units. There is no optimization for specific architectures except the support for high parallelism. This is done intentionally to avoid over optimization for one computing unit while decreasing performance on others. We expect the OpenCL drivers to apply automatic optimization for their platforms during the compile step. Heimel et al. [8] compared their non-optimized OpenCL operators with native written code, showing a similar performance on multi-core CPUs.

Evaluation on TPC-H Query 3

Our first test query is the TPC-H query 3. For our setup, the execution plan includes three selections, three sortings, two indexed-nested-loop joins, and one aggregation. The query plan is shown in Fig. 3. We evaluated the query with three different scaling factors, namely SF 0.1 (100 MB), SF 1 (1 GB), and SF 10 (10 GB). Figure 3a illustrates the placement for the SF 0.1. With only small amounts of data, the operator execution is very short, which results in most computation being placed on the CPU, where the full utilization point and the overheads are low. The joins are compute-intensive through index probing, leading to the placement on the iGPU. Only the sorting of the largest intermediate result can be placed on the dGPU, since the smaller sortings are too fast for the dGPU overheads. This changes with increasing sizes for SF 1 (Fig. 3b) and SF 10 (Fig. 3c). For the latter, all operators are placed on either iGPU or dGPU because the larger data size leads to highly parallel computation and longer-running operations, compensating GPU overheads. The total query runtime for all three scale factors can be seen in Table 2a. We compare the execution of the whole query on a single computing unit to our HOP approach. Changing the execution of the whole query from CPU to the dGPU, a speedup of 1.4x to 2.2x is achieved. Applying HOP decisions, we achieve speedups between 2.6x to 4x compared to the CPU.

Evaluation on TPC-H Query 5

The second evaluation query is TPC-H query 5. The query includes two selections, two nested-loop joins, four sortings,

Table 2 Query run-times (sec) for single computing units and for heterogeneous execution using our HOP approach

	SF 0.1	SF 1	SF 10
(a) TPC-H Query 3			
CPU	0.065	0.587	6.116
iGPU	0.063	0.463	3.842
dGPU	0.047	0.311	2.828
HOP	0.025	0.154	1.524
(b) TPC-H Query 5			
CPU	0.46	4.57	49.08
iGPU	0.31	2.61	21.14
dGPU	0.22	1.32	11.93
HOP	0.13	0.83	7.72
(c) TPC-H Query 6			
CPU	0.003	0.025	0.252
iGPU	0.006	0.031	0.403
dGPU	0.014	0.126	1.247
HOP	0.003	0.019	0.158

three indexed-nested-loop joins, and one aggregation. The nested-loop joins are used for the smallest joins where only 5 or 25 entries are in the smaller input relation of the join. Indexing these smaller relations and applying binary search is not as efficient as plain scanning. As shown in Table 2b, we achieve speedups between 2.1x and 3.5x for using the best computing unit for the whole query instead of the CPU. Our HOP approach achieves 3.5x to 6.5x by placing each operator on the best computing unit instead of using only the CPU. The speedup is higher than the previous query because we have more operators with different sizes and execution behavior so that heterogeneous placement can improve the total runtime immensely.

Evaluation on TPC-H Query 6

The third query, used for our evaluation, is TPC-H query 6. This query only includes three selections and one aggregation plus an intersection of the three independent selection results. This query is slightly different to the other two, since no high computation operators like sorting or joining are needed. As presented in Table 2c, no speedup can be gained by placing the whole query on a different computing unit than the CPU. However, on the fine-grained level of operator placement, speedups of up to 1.6x are achieved by placing the selections and the intersection on the iGPU for larger scale factors. Note that our HOP approach does not need to place the operators heterogeneously. For SF 0.1, all operators are placed on the CPU. While there is no speedup in that placement, the HOP decisions still reflect the best configuration for the given workload.

Evaluation Summary

In our evaluation, we presented our HOP approach for a set of TPC-H queries. We found that, using all computing units in a heterogeneous environment, can outperform a multi-core CPU in cases with large queries or large data sets. In addition, heterogeneity-aware operator placement is more promising in performance than pure whole query placement in heterogeneous environments. Our decisions of individually placing query operators on runtime yields speedups of up to 6.5x in our test scenario.

6 Lessons Learned

In the last sections, we have shown the quality of our proposed HOP approach. With our approach, we achieved high speedups solely through placement decisions instead of executing all operators on the CPU. During our work, we made several observations concerning further optimization of our approach and heterogeneous execution in general.

Parallelism Between Computing Units

The speedups shown in Sect. 5 are achieved solely through operator placement. There is no parallelism between the computing units even if it would be possible for independent operators. We chose this deliberately to present the quality and impact of the placement decisions, which is the focus of this paper. However, we did some side experiments on inter-computing-unit parallelism inspired by the TPC-H query 6. There, multiple selections are independent of each other and can be executed in parallel on different computing units. We change the setup to 20 selections placed in different portions on either CPU and iGPU or CPU and dGPU. The result is illustrated in Fig. 4. We made two interesting observations during our experiments:

1. The CPU/iGPU combination shows much worse performance than the expected speedup through parallel execution.
2. The CPU/dGPU combination is close to the expected performance but shows no significant speedup for the whole execution.

We found that the first observation is caused by the deep integration of the iGPU. The iGPU shares resources with the CPU, especially the same channels to the main memory and the same power envelope. When both computing units are used in peak-load situations, the memory bus and the power supply become bottlenecks. The latter is managed through automatic adjustment of the frequencies of CPU and iGPU. It does not seem possible that both run on their highest frequency at the same time. Stealing each others' resources results in a slowdown of either the CPU execution or the GPU execution or both, leading to this large gap between theoretic performance and real measurements (e.g., Fig. 4a at task placement 8/12).

The second observation concerns the dGPU. Generally, the dGPU does not suffer the same resource-sharing pro-

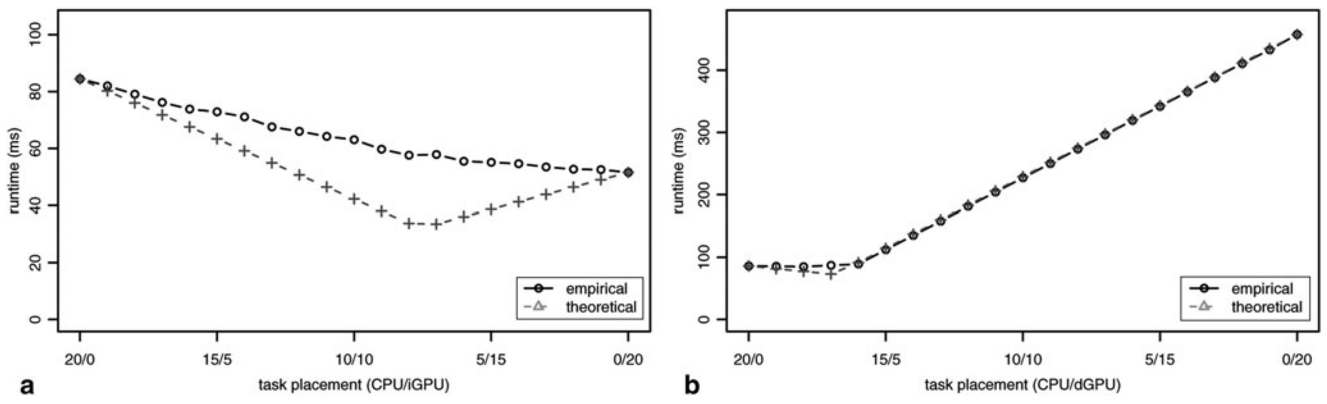

Fig. 4 Evaluation of parallelism between computing units by dividing 20 selection operations on two computing units. **a** CPU and iGPU **b** CPU and dGPU

blems as the iGPU since it has its own memory and power envelope. However, in our example, the combined time of transfer and execution is much higher than that of the CPU, leading to an only small possible speedup through parallel execution. In our experiment, this small potential speedup was reduced through scheduling overhead to nearly no speedup at all. However, if sufficient independent tasks are available or if the HOP model suggests that independent operators are scheduled on the dGPU and an other computing unit, than parallel execution could be beneficial.

No One-Size-Fits-All Approach

For our evaluation, we chose the most promising parallel implementation for each operator. Besides simplicity, this has the effect of similar scaling on each of the computing units, making the model more resistant to wrong estimations. For example, when operator data have much more duplicates or a higher selectivity than previous executions, the estimations will be off and the real execution might be multiple times slower than estimated. However, with the same implementation for all computing units, we can assume that the execution will be multiple times slower for each computing unit. This leads to an optimal or near optimal placement, even with temporarily wrong estimations.

On the other side, one single implementation might not be optimal for all execution scenarios or for each computing unit. The first is already known for database systems where multiple physical operators can be chosen for each logical operator, according to data sizes or selectivity. The latter was also considered in the past, e.g., by using the preferred memory access pattern for each computing unit [8].

To evaluate the need of multiple operator implementation support for each computing unit, we used the sorting operator in different physical implementations. We used a multi-threaded bitonic sort (mt), a multi-threaded radix sort

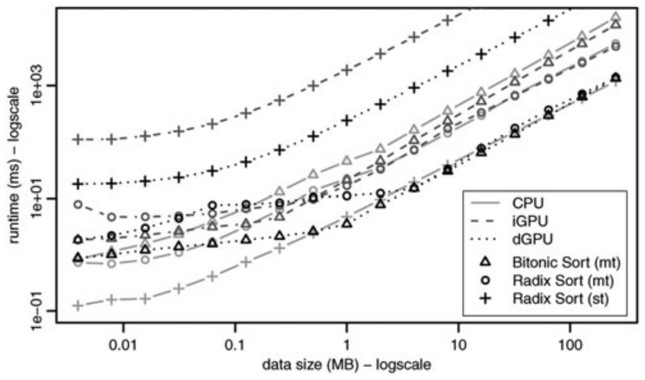

Fig. 5 Evaluation of different sorting algorithms on different computing units

(mt), and a single-threaded radix sort (st). All physical implementations are implemented in OpenCL. The difference between the radix sorts are the degree of parallelism and the overhead of managing intermediate results for the parallel execution.

The results are shown in Fig. 5. On the GPUs, the parallel implementations perform best whereas, on the dGPU, the bitonic sort is almost always better than the radix sort. On the iGPU, the bitonic sort is better for small data sizes and the radix sort shows better performance for large data sizes. However, on the CPU, the single-threaded radix sort always shows the best performance.

The results depend heavily on the amount of processing cores and the memory access bandwidth. When designing a database system for an arbitrary hardware environment, all implementations need to be considered for all problem sizes. Additionally, the amount of implementations could increase even further when supporting multiple programming languages like C++, OpenCL, and CUDA. Handling the diversity of implementations with our HOP approach is generally possible when the execution estimation has enough information but, when relying on heuristic methods, some extensions are needed to handle so many options.

7 Discussion of Future Work

In this section we briefly discuss ideas for future work and their applicability.

The decision model could optimize towards *hiding memory transfers by overlapping* them with the computation of other operators. However, this is only possible if the operators do not depend on each other's results, which is limited within a single query. For multiple simultaneous queries, hiding transfers should be more practical but hard to predict for the decision model. We propose that the model should always consider the transfer costs and the execution engine should try to avoid them while applying the models' decision.

Many different scenarios of *parallelism between computing units* are thinkable, for example (i) running two independent operators at the same time, (ii) dividing the data and running fractions of the same operator on each computing unit, or (iii) running the same operator on each computing unit and proceeding with the result of the fastest. For all cases of parallelism, our observations presented in Sect. 6 need be considered: not every computing unit is beneficial when used in parallel. Additionally, case (ii) introduces new overheads for synchronizing the execution as well as for dividing and combining data, while case (iii) would introduce data copies to all computing units. Since memory transfer is an important factor, we believe that case (i) is most promising for current systems.

To allow global optimization, it is possible to make the *placement decisions on compile time*, e.g., during query optimization. There, it would be possible to avoid unnecessary copy operations and change the entire plan structure for a better heterogeneous execution (e.g., different results for join enumeration). It is open, how to tackle the large search space or even if there is a better plan structure. Current database systems optimize for small intermediate results (especially in join enumeration) and exactly these results build the main part of data transfers between computing units. So the small data sizes are highly beneficial for heterogeneous query execution.

Finally, a system could support *multiple implementations* of an operator optimized for different hardware architectures. Then, the decision model has to find the best operator implementation for each computing unit under the given input parameters. In any case, a fall-back hardware-oblivious version needs to be provided. Furthermore, due to the multiple operator implementations, the whole system implementation and maintenance would become much harder, however, allowing higher performance on computing units that are targeted by the optimized implementations.

All these approaches need to be implemented and integrated into a database system to evaluate their real impact on performance.

8 Conclusion

In this paper, we proposed our Heterogeneity-aware Operator Placement (HOP) model to find an optimal placement of query operators in heterogeneous environments. Our cost model estimates the runtime per operator and computing unit, by using our operator execution model, consisting of latency, data transfer, and execution estimation combined with placement heuristics. Besides reasoning about placement time and placement object, we presented the model in detail together with ways to determine the model parameters. We evaluated our approach on full TPC-H queries within a prototype database engine achieving speedups of up to 6.5x compared to the CPU, solely by adjusting the placement according to our model. Finally, we used our observations to point out limitations and challenges concerning our HOP model and heterogeneous execution in general.

Acknowledgments This work is partly funded by the German Research Foundation (DFG) within the Cluster of Excellence "Center for Advancing Electronics Dresden" and by the European Union together with the Free State of Saxony through the ESF young researcher group "IMData" 100098198. Parts of the evaluation hardware were generously provided by Dresden CUDA Center of Excellence.

References

1. AMD OpenCL SDK and Samples. http://developer.amd.com tools/-and-sdks/heterogeneous-computing
2. Augonnet C, Thibault S, Namyst R, Wacrenier P.-A. (2011) Starpu: a unified platform for task scheduling on heterogeneous multicore architectures. Concurr Comput : Pract Exper 23(2):187–198
3. Boncz PA, Kersten ML, Manegold S (2008) Breaking the memory wall in monetdb. Commun ACM 51(12):77–85
4. 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
5. Govindaraju NK, Lloyd B, Wang W, Lin M, Manocha D. Fast computation of database operations using graphics processors. SIGMOD '04, pages 215–226, New York, NY, USA, 2004. ACM
6. He B, Lu M, Yang K, Fang R, Govindaraju NK, Luo Q, Sander PV (2009) Relational query coprocessing on graphics processors. ACM Trans Database Syst 34(4):21:1–21
7. He J, Lu M, He B (2013) Revisiting co-processing for hash joins on the coupled cpu-gpu architecture. PVLDB 6(10):889–900
8. Heimel M, Saecker M, Pirk H, Manegold S, Markl V (2013) Hardware-oblivious parallelism for in-memory column-stores. PVLDB 6(9):709–720

9. Kicherer M, Buchty R, Karl W (2011) Cost-aware function migration in heterogeneous systems. HiPEAC '11, pages 137–145, New York, NY, USA, 2011. ACM
10. Manegold S, Boncz P, Kersten ML (2002) Generic database cost models for hierarchical memory systems. VLDB '02, pages 191–202. VLDB Endowment
11. Mueller R, Teubner J, Alonso G (2009) Streams on wires: a query compiler for fpgas. Proc VLDB Endow 2(1):229–240
12. Stonebraker M, Abadi DJ, Batkin A, Chen X, Cherniack M, Ferreira M, Lau E, Lin A, Madden S, O'Neil E, O'Neil P, Rasin A, Tran N, Zdonik S (2005) C-store: a column-oriented dbms. VLDB '05, pages 553–564