Adaptive Work Placement for Query Processing on Heterogeneous Computing Resources

Tomas Karnagel, Dirk Habich, Wolfgang Lehner
Database Systems Group
Technische Universität Dresden
Dresden, Germany
{first.last} @tu-dresden.de

ABSTRACT

The hardware landscape is currently changing from homogeneous multi-core systems towards heterogeneous systems with many different computing units, each with their own characteristics. This trend is a great opportunity for database systems to increase the overall performance if the heterogeneous resources can be utilized efficiently. To achieve this, the main challenge is to place the right work on the right computing unit. Current approaches tackling this placement for query processing assume that data cardinalities of intermediate results can be correctly estimated. However, this assumption does not hold for complex queries. To overcome this problem, we propose an adaptive placement approach being independent of cardinality estimation of intermediate results. Our approach is incorporated in a novel adaptive placement sequence. Additionally, we implement our approach as an extensible virtualization layer, to demonstrate the broad applicability with multiple database systems. In our evaluation, we clearly show that our approach significantly improves OLAP query processing on heterogeneous hardware, while being adaptive enough to react to changing cardinalities of intermediate query results.

1. INTRODUCTION

In the last years, hardware changes shaped the database system architecture by moving from sequential processing to parallel multi-core execution and from disk-centric systems to in-memory systems [1]. At the moment, hardware is changing again from homogeneous CPU systems towards heterogeneous systems with different computing units (CUs), mainly to overcome physical limits of homogeneous systems [10]. The computing resources in a heterogeneous system usually have different architectures for different use-cases. Obviously, database management systems (DBMS) need to adapt to this hardware trend to efficiently utilize the given opportunities.

A huge variety of research work has been porting single physical query operators to different processors or CUs like GPUs [12], the Xeon Phi [16], and FPGAs [23], showing great results for isolated workloads. However, for arbitrary queries, we can not expect to execute an operator always on a predefined CU, because depending on data sizes, the data transfers might be more dominant than execution savings. Thus, the placement of physical operators to CUs has to be performed dynamically based on query properties and in particular on characteristics of intermediate results.

Current state-of-the-art approaches for this *placement optimization* use runtime estimation models based on online learning [6, 18] together with global placement optimization at query compile time [5, 11, 17]. For each physical operator within a query and for each available CU, the operator runtime and the data transfer costs are estimated and compared, with the estimations being based on data cardinalities. Up to now, the available approaches [5, 11] assume perfect cardinality estimations even for intermediate results, while this is simply not possible for complex workloads [20]. Even small deviations in the cardinality estimation may have a major impact on the estimated runtime, potentially leading to sub-optimal decisions at the end (error propagation). Moreover, we identified two additional limiting aspects of state-of-the-art approaches. First, the execution runtime of the same physical operator can behave differently depending on the query structure and the input data size, potentially resulting in imprecise runtime estimation on operator level. Second, a decision taken regarding the location of intermediate results in a heterogeneous computing environment limits the flexibility for future placement decisions within a physical query plan, due to dominant data transfer costs.

To overcome these limitations, we propose a novel adaptive placement approach for query processing on heterogeneous computing resources, which is intentionally not part of the query optimizer phase. Our approach takes a physical query execution plan as input and divides the plan into disjoint *execution islands* at compile-time. The *execution islands* are determined in a way that the cardinalities of intermediate results within each island are known or can be precisely calculated. The placement optimization and execution is performed separately per island at query runtime. The processing of the *execution islands* takes place successively following data dependencies. To further enhance our approach, we propose two additional improvements: (1) a fine-grained runtime estimation technique and (2) a placement-friendly data transfer technique. Moreover, our developed concepts are combined with traditional approaches to an *adaptive placement sequence* defining the optimal application order of the different techniques. We

Proceedings of the VLDB Endowment, Vol. 10, No. 7
Copyright 2017 VLDB Endowment 2150-8097/17/03.

also provide an implementation of our sequence as a virtualization layer called HERO (HEterogeneous Resource Optimizer). We evaluate our approach using synthetic benchmarks and use two different database systems to present the real adaptive behavior and the performance of our approach. In detail, our contributions are:

- Pointing out the problems and dependencies of state-of-the-art operator placement approaches for heterogeneous computing resources. (Section 2)
- Proposing our adaptive placement approach, which is independent of cardinality estimations implying improved runtime estimations and better placement decisions. (Section 3)
- Defining an adaptive placement sequence for heterogeneous hardware environments. (Section 4)
- Outlining the prototypical implementation of our approach as a virtualization layer, allowing the evaluation on a range of existing database systems. (Section 5)
- Presenting an exhaustive evaluation including speedups of 50x by efficiently using heterogeneous hardware, while being adaptive to changing intermediate cardinality. (Section 6)

Finally, we conclude the paper with related work and a summary including future work in Section 7 and Section 8.

2. PLACEMENT OPTIMIZATION

Heterogeneous hardware usually consists of many different CUs, like CPUs and GPUs, each with individual properties making them useful for different tasks. DBMSs running on such hardware commonly exploit a block-wise execution model, like vector-at-a-time or column-at-a-time model for query execution [5, 11, 14, 22, 28]. Thereby, the physical query operators are either implemented in OpenCL [14, 28], allowing them to be executed on different CUs, or provide separate implementations for CPUs and GPUs using C++ and CUDA [5, 11, 22]. Most systems are optimized for single query latency [11, 14, 28]. Therefore, our approach is based on the same architecture.

To demonstrate the feasibility and wide applicability of our adaptive placement approach, we work with two different DBMSs in this paper: gpuDB [28], a newly implemented prototypical system, and Ocelot [14], an extension of MonetDB [3]. Both DBMSs are based on OpenCL, using OpenCL kernels to execute a query according to the column-at-a-time model with full materialization of intermediate results. It is important to note, gpuDB mainly supports the Star Schema Benchmark (SSB) [25], while Ocelot has been evaluated using the TPC-H benchmark.

2.1 Placement Description

To compute query results, DBMSs traditionally compile SQL queries into query execution plans (QEP), whereas the *query optimizer* applies logical and physical optimizations to determine the most efficient QEP. For systems running on heterogeneous computing resources, an additional *placement optimization* is performed before execution [5, 11]. The main objective of this placement optimization is to assign physical operators of the most efficient QEP to the ideal CUs. To achieve this, query properties need to be considered, including QEP structure, data cardinalities and characteristics as

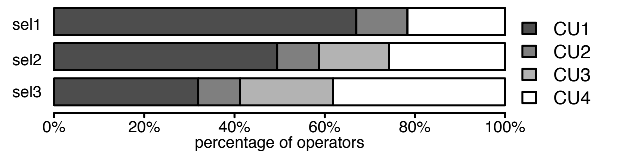


Figure 1: One query with different (intermediate) cardinalities. (SSB query 3_4, sel3 $\approx$ 2$*$sel2 $\approx$ 6$*$sel1)

well as the capabilities of each single CU, including execution time and data transfer costs.

Generally, this *placement optimization* is based on mathematical runtime estimation models (for execution and data transfer), where each possible mapping of physical operators to CUs is evaluated and the placement with the lowest estimated overall runtime is chosen. As a matter of fact, the decision largely depends on the processed and transferred data in terms of data cardinalities. To determine the most-efficient placement, this optimization has to find a trade-off between optimal execution (every operator on its preferred CU) and reducing data transfer costs (ideally single-CU execution). After this *placement optimization*, the most efficient QEP is executed according to the determined placement.

2.2 Open Challenges

While evaluating the current state-of-the-art in this domain, we identified three open challenges $C1$ to $C3$.

C1 - Inaccurate Cardinality Information

Generally, data cardinality information is influencing the estimation models for the traditional query optimization as well as for the placement optimization. This information is usually provided via statistics, histograms, or estimations using heuristics. However, especially when working with many joins, groupings, or complex selections, the estimated cardinalities for intermediate results can show significant errors [15]. Selected attributes can be correlated and statistics on data distribution can not be simply intersected for different attributes or relations [8]. Leis et al. report cardinality estimation errors by a factor of 1,000[1] or more for all tested DBMSs when the query has multiple joins [20].

To demonstrate the high influence of cardinality information on the *placement optimization*, we execute a single SSB-query with three different selectivities, resulting in three different intermediate cardinalities. More details about the experiment can be found in Section 6.5. Figure 1 shows the optimal placement distribution of the query operators to four different CUs. As we can see, the ideal placements vary greatly, which is mainly caused by different intermediate cardinalities. This illustrates the importance of cardinality information for the *placement optimization*. Unfortunately, existing approaches only assume exact knowledge of data cardinalities for intermediate results [11, 5], but they obtain the information from the query optimizer. These approaches ignore the well-known problem of inaccurate cardinality estimation, which might result in sub-optimal placements.

C2 - Inaccurate Runtime Estimation

To assign operators to CUs, the runtimes of operators have to be estimated. Various approaches have been proposed

[1]A factor of 1 is accurate, while 10 means ten times more or less.

including estimation based on instruction basis [13], basing the runtime estimation on small benchmarks [11], or monitoring and learning of the execution behavior in an online manner [6]. Especially, the learning approach is promising for complex operators and different CU architectures. However, learning-based approaches suffer from inaccuracy originating again from wrong cardinality information as well as behavior changes in the operator. We observed the latter by experimentally investigating our two foundational DBMSs Ocelot [14] and gpuDB [28]. There, the same physical operator behaves differently depending on input data size or the position within the QEP. The reason for these variations are pre-processing steps, like bitmap materializations or hash table creations, as well as post-processing steps, like bitmap concatenations or data conversions. The presence or absence of these extra steps is usually not visible in a purely operator-based query execution plan, however, these additional steps do influence the runtime of the operator.

C3 - Influence of Intermediate Result Location

During query processing, operators should be executed on different CUs, with input data being transferred to, and stored in the CU's memory together with the operator's results. With intermediate results being stored on a specific CU, the further processing is usually locked to this CU, even if other CUs perform better. The reason for that are the transfer costs, which might be dominating the query runtime. As a consequence, current approaches for placement optimization are substantially dominated by data transfer costs rather than optimal operator execution limiting the usage of heterogeneous computing resources for a single query.

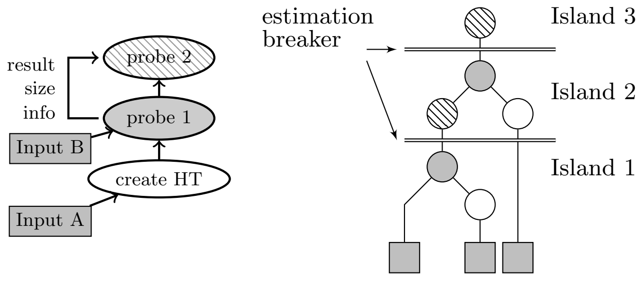
3. ADAPTIVE PLACEMENT OVERVIEW

As mentioned in the previous section, unknown or wrongly estimated cardinalities are the most significant source for errors during *placement optimization* (challenge $C1$). In our novel adaptive placement approach, we do not strive to improve the cardinality estimation in general, but focus on becoming completely independent of these estimations. To tackle the remaining two challenges, we additionally introduce a fine-grained runtime estimation and a less data-centric optimization to improve the placement quality of our approach.

3.1 Adaptive Placement Approach

Our approach to tackle challenge $C1$ is two-fold: (1) creating *execution islands* at compile-time and (2) applying placement optimization and execution per island at run-time.

At **query compile-time**, the query optimizer provides the most-efficient QEP as input for our placement optimization, as done by all state-of-the-art approaches. Then, we divide the QEP into disjoint *execution islands*, which combine subsequent operators of the QEP in a way that within a single *execution island* the cardinalities of intermediate results are known or can be precisely calculated at run-time. The islands are delimited by so called *estimation breakers* defining the QEP positions, where new cardinality information will be available during processing. At **query runtime**, the disjoint *execution islands* of the QEP are executed successively. Before executing the operators within an island, placement optimization for this island is conducted at run-time. Since we know the exact cardinalities within this *execution island*, we are able to precisely estimate the



(a) Hash join implementation. **(b) Execution islands created by the unknown join result size.**

Figure 2: Highly parallel hash join processing.

runtime behavior of each operator as well as data transfer costs. Thus, our placement decisions are done on accurate numbers. To make these decisions, we propose *regional* optimization, which is essentially global placement optimization restricted to a specific execution island. Whenever the execution of an island is finished, we reach an estimation breaker and the intermediate cardinalities for the next island can be calculated.

The most challenging issue for our approach is the division of the QEP by the identification of *estimation breakers.* To determine these *estimation breakers*, we analyzed the execution behavior of physical operators in the underlying DBMS. Since almost all CUs in a heterogeneous hardware system offer high parallelism, this aspect also affects the implementation of operators. For example, when thousands of threads work on the same data, traditional locks or atomic operations hinder parallelism and concurrency significantly. However, if the result cardinality is precisely known, each thread can compute the designated position of its output and execute its work without locking or synchronization.

To achieve this desired processing behavior, the exact cardinalities are usually computed within a probing step first, before producing the actual operator results. Figure 2a illustrates that approach for a hash join operator, as proposed by He et al. [12]. A traditional hash join consists of two steps: (1) hash table creation and (2) hash table probing. A highly parallel version has two probing steps. The first probe calculates the output size for each thread, where the actual size of this probe's output is exactly one value per thread. Hence, the output size of the first probe is known beforehand. Afterwards, the second probe uses the gathered cardinality information, knowing precisely the real output size, and produces the actual join result in parallel.

Our QEP division into *execution islands* is based exactly on that execution observation. We analyze the QEP and the corresponding operators at compile-time to determine such cardinality probing steps within the operators. These probing steps are our *estimation breakers*, because new cardinality information is becoming available. For example in Figure 2a, *Probe 1* computes the result cardinality of *Probe 2*, therefore an *estimation breaker* would be added between both and the operator placement would be optimized for two separate islands. A larger example is given in Figure 2b showing a query with two hash joins, where the two probing steps of each join divide the query into execution islands. For the first island, the intermediate cardinalities are known, based on the cardinalities of the input tables. For all parts within this island, the placement can be defined using runtime estimation and placement optimization, based on exact cardinality knowledge. Afterwards, the island is executed according to the chosen placements. Once the island is fully executed, the intermediate cardinalities for the next island can be calculated (e.g., after *Probe 1*) and we can start the same process again until all islands are executed.

To summarize, through our adaptive approach, the estimation of operator runtimes and data transfers works on precise cardinality information, significantly improving both estimation results.

3.2 Improving the Placement Quality

With the previous approach, we are independent of cardinality estimations of intermediate results. In addition to that, we propose two further techniques in order to improve our placement quality according to challenges $C2$ and $C3$.

Fine-grained Runtime Estimation ($C2$)

To improve runtime estimation and better support our adaptive placement approach, we propose to work on sub-operator granularity, where sub-operators can be reoccurring functions that are executed subsequently within an operator. Specific pre-processing and post-processing steps can be expressed with specific sub-operators, allowing the estimator to consider their runtime individually. Working on this fine-grained level has several advantages:

1. More accurate runtime estimations, since every processing step is considered separately.
2. More training data, as same sub-operators can be used in different operators (e.g., the same hash table creation for hash joins and hash-based groupings).
3. More fine-grained placement, as sub-operators can be placed separately, instead of placing full operators.
4. Support for our adaptive work placement, as sub-operators allow the positioning of *estimation breakers* (see example in Figure 2b).

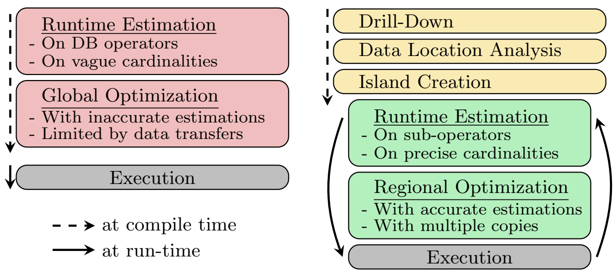
While the first two points improve the runtime estimation quality, the third point could potentially improve the runtime of the whole query by fine-grained placement decisions.

Intermediate Results with multiple Locations ($C3$)

To provide the optimizer with more freedom in choosing the best CU, we propose keeping temporary copies of used data on the CUs. Data objects can be accessed and updated by sub-operators, however, they have to be transferred to and from the CUs depending on the access location. Instead of moving a data object to the CU, where it is used next, we copy the data, while also keeping the original version. This enables two improvements to the placement optimization:

1. Future executions can choose a CU, where a copy of needed data is stored and, therefore, avoid additional transfers. Otherwise, they can choose different CUs, transfer data, and provide another copy to future executions.
2. Parallel accesses to different copies on different CUs is made possible, while before accesses on the same data object had to be sequential.

Allowing copied data objects introduces challenges with consistency, when data is updated. To address this problem, we define a small set of rules for data handling: (1) Is the needed data object not on the assigned CU, copy it from a different CU where it resides. If multiple CUs apply (e.g., multiple copies as source), use the CU with the smallest transfer costs. (2) If a sub-operator is updating data, delete all copies except the one used by the sub-operator. (3) If CU's memory space is full, delete older copies.



(a) Traditional Approach **(b) Adaptive Approach**

Figure 3: Novel Optimization Sequence: preprocessing (yellow) extends traditional steps (green).

When the majority of memory accesses is read-only, this approach can lead to temporary copies being on nearly every CU. There are no additional transfer costs for this approach, as the system would transfer data objects with or without copy support. However, when using copies, data is available on the source and the target after the transfer, potentially avoiding future transfers. All copies can be removed with query termination.

4. ADAPTIVE PLACEMENT SEQUENCE

In this section, we provide more insights to our adaptive approach by defining an optimal execution sequence of the proposed techniques as methodology for our placement optimization. Figure 3 compares the traditional approach with our adaptive approach. As illustrated, our adaptive placement sequence has five optimization steps, where the first three steps act as preprocessing at query compile-time. Then, the following two steps and the actual query execution are applied multiple times, once per execution island, at run-time. The steps are illustrated using the following example query based on the TPC-H schema:

SELECT sum(l_quantity) FROM lineitem
WHERE l_discount < 20 and l_quantity < 24

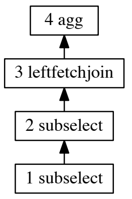
When executing this query for example with Ocelot [14], a QEP with four operators is generated, as shown in Figure 4a. The first two operators are the selections on *l_quantity* and *l_discount*, both producing bitmaps. The third operator materializes the result of the selections before the fourth operator performs the aggregation. All four operators are implemented in OpenCL, so they can be executed on different CUs without any code adjustments.

4.1 Steps at Query Compile-Time

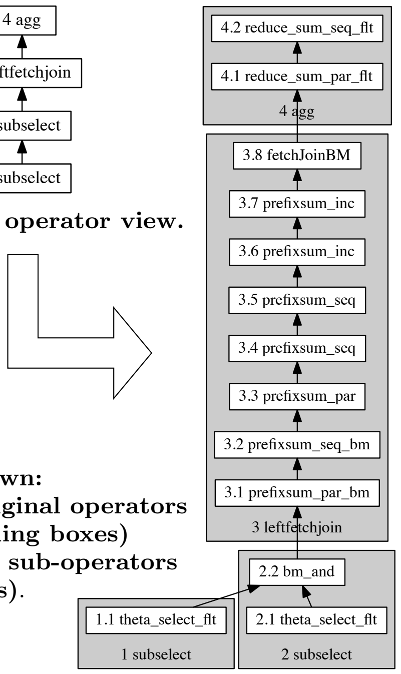
At query compile-time, we divide the most-efficient QEP determined by the traditional query optimizer into *execution islands* with the following three steps.

4.1.1 Drill-Down

As motivated before, our overall approach works on sub-operator level to be able to determine *estimation breakers* within operators as well as to improve the placement quality. Therefore, our first preprocessing step is a *Drill-Down* from operators to sub-operators of the QEP.



(a) Original operator view.



(b) Drill-Down: The four original operators (gray bounding boxes) split into 13 sub-operators (white boxes).

Figure 4: Operator and sub-operator view; numbers define the execution order; arrows symbolize the data flow.

Example: In our Ocelot example, sub-operators correspond to OpenCL kernels. Figure 4b shows the drill-down result for our running example. *Theta_select_flt* performs a selection on float values, while *bm_and* intersects the two bitmaps of the previous selections to one resulting bitmap. The *prefixsum_** sub-operators count the number of set bits within the resulting bitmap in parallel to calculate the result cardinality. The *fetchJoinBM* sub-operator materializes the bitmap result while fetching values of *l_quantity*, as they are needed for the next steps. Finally, the *reduce_sum_** sub-operators consume the materialized selection result to build the aggregation sum in parallel.

From a high-level point of view, both selections are assumed to have equal runtime, because both consume a column of the same table and work with the same data cardinalities for input and output. However, as we see in Figure 4, the two operators differ in execution on the fine-grained level. Both selections execute a *theta_select_flt* sub-operator but the second selection has an additional sub-operator to combine the two bitmaps, increasing the runtime of the second operator. When working directly on sub-operators, the estimator learns the runtimes for *theta_select_flt* and *bm_and* separately, leading to more precise runtime estimations and also potentially different placement decisions for these sub-operators. Additionally, we can not assume that a single placement decision is optimal for all eight sub-operators of the *leftfetchjoin*. When placing only operators, we loose speedup opportunities if sub-operators are diverse and run better on different CUs.

4.1.2 Data Location Analysis

After our *Drill-Down*, we analyze the resulting data flow of the query plan and in particular the memory accesses of the sub-operators (read and write). This analysis is done to determine where intermediate results can be kept in different locations according to our temporary copy concept for challenge *C*3. Please note, we only find opportunities for copies in this second step, while the real occurrence of copies is placement dependent. For example, if two sub-operators only read the same data, our analysis confirms that copies may be kept. However, if both sub-operators are placed on the same CU, there is no data transfer and, hence, no additional copy.

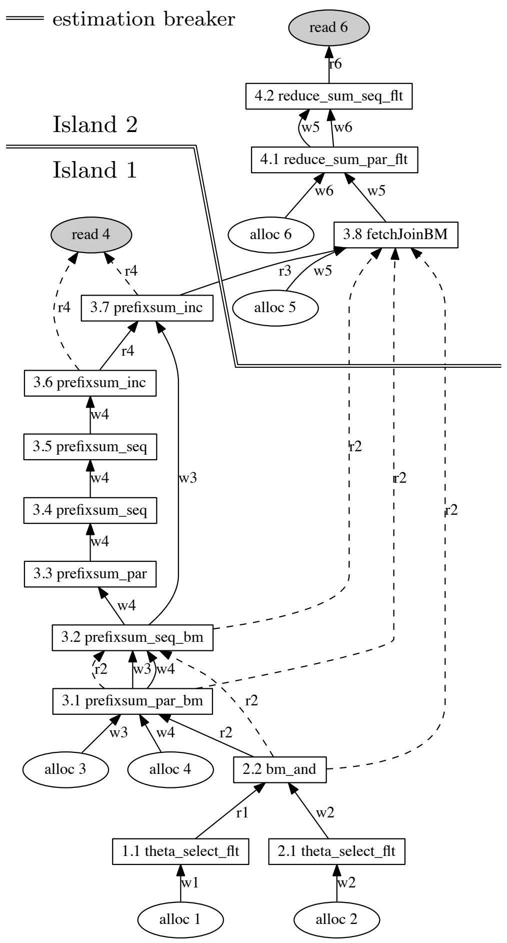


Figure 5: Reordering data accesses by inspecting data objects (ovals) and dependencies. The data accesses are either read (r) or write (w) on a specific data object. Dashed lines illustrate that there is a choice of data source.

Example: The benefits are illustrated in Figure 5. There, we add the memory operations *alloc*, *read*, and the memory access types for each sub-operator (*r* for read-only, *w* for read-write). *Alloc* allocates the data object, while *read* evaluates a result. The first read on *data object 4* evaluates the result size of the bitmap materialization, which determines the size of *data object 5* (hence, the estimation breaker). The last read is done to output the aggregation result. One advantage of our analysis can be seen for *sub-operator 3.8*. It reads *data object 2*, which was used before by *sub-operators 2.2, 3.1*, and *3.2*. If their placements were on three different CUs, *sub-operator 3.8* can now choose one of the three CUs, where a copy resides, while having no transfer costs for this data. An additional advantage can be seen for the first read operation. There, it was scheduled to be executed after *sub-operator 3.7*. However, since both operations only read the data, they can be executed in parallel using copies.

4.1.3 Island Construction

Our third pre-processing step is the island construction by traversing the data flow, collecting all sub-operators with predefined input and result cardinalities into a single execution island, and creating a new island after an estimation breaker. For example, selections producing bitmaps, sort operations, foreign key joins, calculations, and aggregations produce fixed size results, and may be executed within the same island. However, for bitmap materializations, groupings, and joins not based on foreign keys, the cardinalities are not fixed, leading to a staged execution: (Phase 1) calculating the result size and (Phase 2) actually producing the result. Between these two phases, we place our *estimation breakers* dividing the query plan into *execution islands*. Thus, the number of *execution islands* depends on the used sub-operators within a query.

Example: In Figure 5, the intermediate results are fixed for both selections, as they consume a column of *lineitem* and produce a bitmap with one bit per row. However, when the bitmaps need to be materialized, the exact result cardinality is unknown. This results in an estimation breaker between *sub-operator 3.7 and 3.8*, building two separate execution islands. The first island combines the selections and calculates the materialized result size, while the second island combines the last step of materialization and the aggregation.

4.2 Steps at Query Run-Time

With the presented steps at query compile-time, the most-efficient QEP is divided into *execution islands*. The following two steps of *runtime estimation* and *placement optimization* are directly applied before an island is executed in order to determine the optimal placement for the specific island.

4.2.1 Runtime Estimation per Island

The first step at query run-time is to estimate the runtimes of sub-operators for the available CUs. Here, we utilize an automated black-box online-learning approach [18]. We continuously monitor the execution times together with the input data cardinalities. This has to be collected for all combinations of sub-operators and CUs. The actual runtime estimation for sub-operators uses the given input data cardinality and linear approximation between the two closest collected measurements. Using this black-box approach, we do not require complex behavior modeling, we are not bound to any specific operator implementation, and we can support future hardware without prior knowledge.

Additionally, we also have to estimate the transfer costs. Since transfers are not dependent on queries, operators, or data distributions, we do not need to monitor transfer times at run-time. Instead, we collect the underlying information using a simple benchmark during a ramp-up phase of a system for calibration. Again, we estimate the transfer costs using a linear approximation based on data cardinalities.

Example: A possible output of this step for our running example is depicted in Table 1. Here, we assume to have four different CUs and we assume that the estimation model gives the hypothetical runtimes for each combination of sub-operator and CU. The runtimes are in abstract *units* and chosen in a way that they can demonstrate a specific

Table 1: Hypothetical runtime estimations (in *units*) for the given sub-operators and 4 CUs.

	Sub-operator												
	1.1	2.1	2.2	3.1	3.2	3.3	3.4	3.5	3.6	3.7	3.8	4.1	4.2
CU 1	2	2	4	3	4	3	3	4	1	2	5	3	4
CU 2	4	4	5	5	3	3	1	3	1	4	2	3	1
CU 3	2	2	2	1	5	5	1	1	5	1	1	4	5
CU 4	2	2	4	4	1	1	2	1	3	3	2	3	2

optimization aspect later. Furthermore, we assume any data transfer between two CUs takes one *unit*.

4.2.2 Placement Optimization per Island

To define the actual placement, we apply our *regional* optimization approach. Unlike global optimization, regional optimization is limited to the sub-operators within one execution island to ensure that only exact cardinalities and runtime estimations are used in the decision process. Since the search space is too large to evaluate every possible placement, we apply a light-weight greedy algorithm [17]. The greedy algorithm tries to improve each sub-operator's placement by considering its runtime estimations and the regional context, where data transfers from preceding and to succeeding sub-operators are considered. The algorithm works like the following:

```
placement = starting_placement

repeat –
    for all sub-operators in execution island:
        cost = execution_est(CU)
                + input_transfers_est(CU)
                + output_transfers_est(CU)
        update_placement_to_best_CU(cost)
 – until ( has_not_changed(placement) )

for all sub-operators in execution island:
    execute(placement)
```

Since the result depends on the starting placement of the greedy algorithm, we first run the algorithm starting with each single-CU placement. If the estimated runtime of the best-plan found is larger than a threshold (in our case 100ms), we allow more time for optimization by evaluating multiple random starting placements. The placement with the best estimated runtime is chosen for execution.

Example: In our running example, we want to show the effectiveness of our optimizations and a comparison to state-of-the-art approaches. To illustrate the effectiveness, we consider the placement of operators, of sub-operators, and sub-operators with data copies in Table 2. We also compare our optimization approach to local and global optimization [17]. *Local* optimization decides the placement of each (sub-)operator locally, by evaluating the runtimes of input transfers and execution, while subsequent operators and their data usage is not considered due to the local view. Since the (sub-)operators are directly executed after the placement decision, the data cardinalities are always precisely known. The interplay between local placement and execution works like this:

```
for all sub-operators:
    cost = execution_est(CU) + input_transfer_est(CU)
    execute_sub-operator_on_best_CU(cost)
```

Global optimization is similar to our regional approach, but the placement optimization is done for the full query plan at once. Therefore, data cardinalities for intermediate results are not known at that time and have to be esti-

Table 2: Results of optimization (in *units*) for the given query and runtime estimations. The gray values are visualized in Figure 6.

Optimization	Operator	Sub-Operator	+ copies
Local	30	34	33
Global	29	27	26
Regional	29	28	27

mated for the placement decisions leading to the described drawbacks. For comparability, we use perfect cardinality information for the global optimization in the example.

As we can see in Table 2, *local* placement always yields the worst runtimes, while *global* under the assumption of perfect cardinality information shows the best results. Our *regional island-based* optimization represents a middle path, with a performance close to the *global* strategy.

Generally, placement on operators is usually worse than placement on sub-operators, except for the local strategy, where sub-operator optimization produces additional data transfers. For all other cases, a drill-down to sub-operators is beneficial. Additionally, allowing copies can achieve an improvement through avoided transfers for all optimizations. In this example, the improvement is limited to a maximum of two avoided transfers as there are only two sub-operators able to exploit copied data objects (*sub-operator 3.2 and 3.8*).

Figure 6 depicts the placement decisions for *local*, *global*, and our *regional* optimization when using sub-operators and copies. It shows that *local* optimization is swapping CUs often to achieve better results in execution, while introducing many additional transfers (e.g., sub-operator 3.3 to 3.5). *Regional* and *global* optimization only differ for *operator 3.7*, where global optimization chooses an additional data transfer to avoid transfers for the following operator. Since this is exactly at the border of two islands, the *regional* approach can not optimize this far ahead and chooses a locally better placement.

To summarize, our regional placement is close to the global result. Nevertheless, it is easy to imagine, if perfect cardinality estimations are not available—which is the normal case—, then the global placement becomes worse. However, since our regional approach always works with precise cardinality information at run-time, we are not affected by it. In this case, our regional approach outperforms the global placement optimization as presented in our evaluation.

4.3 Feasibility of our Approach

Our adaptive optimization approach is best applied to columnar in-memory DBMS with a column-at-a-time processing model, where intermediate results are materialized, mainly triggered by related work like Ocelot [14] and gpuDB [28]. However, other execution models may also benefit from heterogeneous execution and our adaptive placement.

Block-wise processing, e.g., as vectors [4] or morsels [19], can be offloaded and placed on different CUs, where different blocks executing the same operator can even run in parallel on different CUs at the same time. If intermediate results are not materialized after every operator (like pipelined execution [4] or generated query code [24]), complete pipelines can be offloaded until a pipeline breaker forces materialization. Although this results in a more coarse-grained placement, multiple pipelines can be grouped to execution islands until an intermediate cardinality needs to be calculated. The main challenge within the pipelined execution is runtime estimation, because pipelines or generated code could always differ in their execution. However, an estimation based on operations within the pipelines is possible.

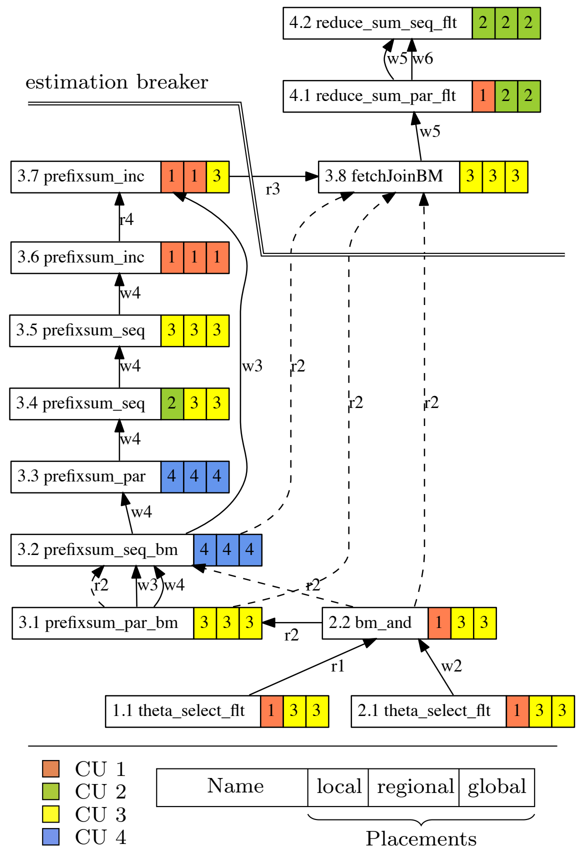


Figure 6: Placements for sub-operators with copies. The colors and numbers represent the placement.

The only processing model that would most likely not benefit from our approach is tuple-at-a-time, which is already not cache-friendly and has a high function-call overhead. These problems would increase in heterogeneous environments, where the communication and data transfer between CUs is costly. Therefore, this processing model is not well-suited for heterogeneous systems in general.

5. IMPLEMENTATION

To broaden the scope of our evaluation as well as to show the wide applicability of our approach, we want to support many heterogeneous DMBSs like Ocelot [14] or gpuDB [28]. Due to implementation efforts, we restrict ourselves to OpenCL-based system offering the opportunity to run on different CUs with a single code basis.

5.1 General Architecture

Each hardware vendor supplies an OpenCL driver to their CUs, which is loaded when the CU is first accessed. The driver manages the communication of the application to the CU using the standardized OpenCL interface. For our objective to support many heterogeneous DBMSs, we prototypically implement our own OpenCL driver called HERO (HEterogeneous Resource Optimizer). Figure 7 shows its architecture. Our OpenCL driver transparently manages the

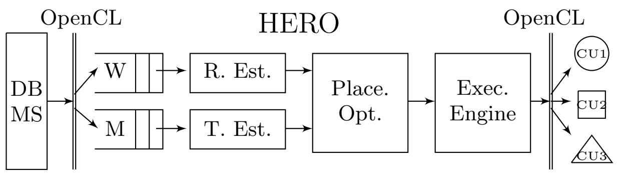


Figure 7: Architecture overview of HERO.

heterogeneous environment and includes our adaptive placement approach. Using this driver, an OpenCL-based DBMS itself needs no or only a few adjustments to utilize our placement strategy. The DBMS then communicates with HERO as with a single CU, while HERO manages all available CUs underneath. HERO does that by intercepting the communication between DBMS and CU, while applying our adaptive placement sequence and executing the work heterogeneously.

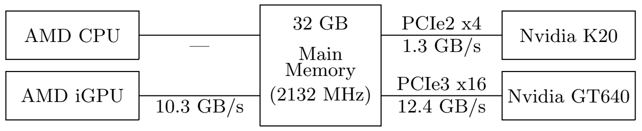
For incoming work, the OpenCL interface itself provides the *Drill-Down*, since only sub-operators (in this case OpenCL kernels) and memory operations can be submitted. Sub-operators are collected in a work queue (W), while all memory accesses are stored in a memory manager (M) to provide the *data location analysis* in a later step. Requesting an intermediate result acts as *estimation breaker*, triggering *island construction* and all further steps. *Runtime estimation* (R. Est.) is done for all sub-operators of an island including the estimation of possible transfer costs (T. Est.). Afterwards the placement optimizer (Place. Opt.) determines the heterogeneous *placement* by using runtime estimations, transfer estimations, the data access analysis, and the island view on the query. The placed sub-operators are then executed using the OpenCL interface and all available CUs.

5.2 Advantages and Limitations

The main advantage of this virtualization is its re-usability for different OpenCL-based database systems. Most of these systems can instantly be transformed from a single-CU execution model to adaptively optimized heterogeneous execution, by simply loading our HERO driver instead of the standard OpenCL drivers. Besides the performance improvement, this allows simpler decoupled development and maintainability, without the additional effort of handling the heterogeneous hardware, transfers between different CUs, runtime estimation, and heterogeneity-aware optimization within the DBMS.

However, there are also limitations to our virtualization. Most importantly, crucial information from the DBMS is missing, like memory access types of operators and sub-operators. As a solution, we partially compile OpenCL sub-operator code at startup time to the LLVM intermediate representation using the Clang compiler. In this compilation process, Clang evaluates the memory access types of given OpenCL kernels to the input data, which we can extract for our data location analysis.

To achieve good performance, OpenCL kernels sometimes need to be optimized for a specific CU. The most common example are memory access patterns, which (in many cases) should be different for CPUs and GPUs [14]. To support such optimizations, we allow different OpenCL kernel variants. The DBMS specifies the default kernel to use, while HERO might exchange that kernel with an architecture-optimized variant, depending on the placement decision.



Name	Model	Memory	#Cores	Frequency	GFLOPS[4]
CPU	AMD A10-7870K	30 GB	4	3900 MHz	44.17
iGPU	AMD Radeon R7	2 GB	512	866 MHz	877.18
K20	Nvidia Tesla K20	5 GB	2496	706 MHz	2900.63
GT640	Nvidia GT 640	2 GB	384	901 MHz	601.97

Figure 8: Heterogeneous evaluation setup consisting of one CPU and three different GPUs.

Another limitation is the use of OpenCL itself, since there are DBMSs, where operators are implemented in different programming languages, not using OpenCL [5, 11]. These operator implementations are not as extensible as OpenCL in respect to different hardware setups, however, they might yield more performance through more specialization. Our adaptive placement sequence is general enough to work for these systems as well. Nevertheless, our current *implementation* is limited to OpenCL. As a conclusion, we use our implementation as a way to evaluate our adaptive placement approach, while the final integration into a productive database system remains future work.

6. EVALUATION

Our evaluation is based on our OpenCL driver implementation with the OpenCL database systems gpuDB[2] [28] and Ocelot[3] [14]. We use gpuDB for a large part, while the portability is shown for Ocelot at the end. Therefore, we describe gpuDB now and Ocelot in Section 6.6.

Generally, gpuDB [28] is a prototypical OpenCL-based query execution engine to process OLAP queries, whereas it mainly supports SSB queries. During an offline compilation step, each query is compiled into a binary. In the original gpuDB version, a query can only run on a single CU, which has to be chosen beforehand. With HERO, we make the query execution truly heterogeneous. We execute gpuDB with the supported 13 SSB queries and a scale factor of 10. Since gpuDB has one binary per query, it is easy to monitor where time is spent, therefore, we choose gpuDB for detailed evaluation.

For all tests, we run each query more than 10 times. We disregard best and worst runtimes to remove outliers and average the remaining query runtimes as the final result.

6.1 Hardware Setup

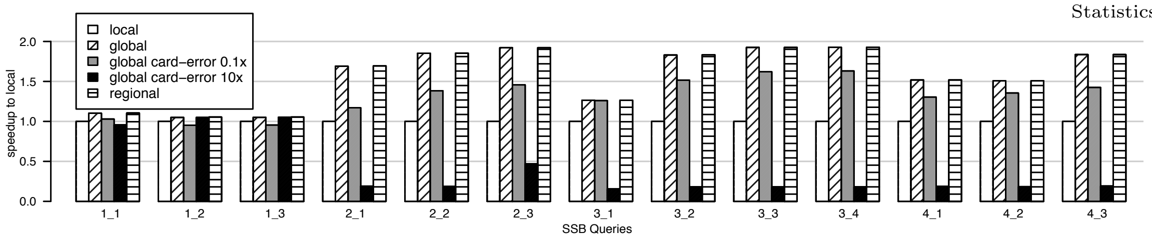
We use a highly heterogeneous hardware setup consisting of a CPU and three different GPUs, because of their general availability and their support for OpenCL. Future systems could also include other accelerators like Xeon Phi. Figure 8 illustrates our system, its connections to the main memory, and an overview of each CU. In the following, we use the names stated in this figure for each CU.

Connections: The CPU and the AMD integrated GPU (iGPU) can access the main memory directly, but the iGPU has a higher memory access bandwidth during execution

[2]gpuDB (commit 609): code.google.com/archive/p/gpudb/

[3]Ocelot (version 1cda9db): bitbucket.org/msaecker/monetdb-opencl

[4]Single precision performance measured with the benchmarking tool *clPeak* (version 898a155): github.com/krrishnarraj/clpeak



Statistics

Q	O	S	I
1_1	9	38	5
1_2	9	39	5
1_3	9	40	5
2_1	18	97	12
2_2	18	100	12
2_3	18	91	12
3_1	20	111	13
3_2	20	109	13
3_3	20	111	13
3_4	20	108	13
4_1	23	135	15
4_2	24	144	15
4_3	24	140	15

Figure 9: Optimizations and statistics on SSB queries (#(O)perator, #(S)ub-operators, #execution (I)slands)

if data was copied to a dedicated part of the main memory before. Therefore, the iGPU has to transfer data first with max 10.3 GB/s, while the CPU can work on data in main memory without any transfers. The presented bandwidths are data transfer bandwidths and not memory access bandwidths. Each CU has a different memory access bandwidth once the data is transferred to its own memory. Both Nvidia GPUs use PCIe, while the GT640 uses the 3rd generation with 16 lanes; the K20 uses the 2nd generation with 4 lanes (our K20 does not support PCIe3). The presented connection bandwidths are max values observed by HERO. Transfers between two CUs have lower bandwidths, since data has to be transferred to main memory first, before being transferred to the next CU. For example, transferring 1GB of data from the K20 to the GT640 can be with 1.18 GB/s.

Performance: We use an OpenCL-based benchmark to exemplarily compare the performance of the CUs (GFLOPS in Figure 8). The performance differs greatly, however, given the connection bandwidths, differences in architecture, and different memory hierarchies on the CU, it is not trivial to find the best CU for a given workload. The shown results for single-precision performance are only indicators of the CUs' performance, while the real performance for a database workload depends on many factors.

6.2 Micro Benchmarks

To show the impact of our adaptive approach, we firstly collect single-CU runtimes of all sub-operators together with information about operators, execution islands, and transfer costs. The information is then used offline by evaluating different estimation and optimization methods based on our HW setup running gpuDB using SSB queries.

Fine-grained optimization: We claim that sub-operator based estimation improves the estimation quality and that placing sub-operators instead of operators can lead to performance improvements. For the evaluation, we insert all collected execution data over all queries into our runtime estimator. Then, we estimate the (single-CU) runtime based on operators and sub-operators and compare the estimation to the real execution time. Operator-based estimation shows a 4.73ms mean absolute error (MAE) over all operators in all queries, while sub-operator estimation has only 0.7ms MAE. Additionally, the maximum errors differ largely from 58ms for operators and 1.3ms for sub-operators. The error difference is mainly caused by same operators containing different amounts of sub-operators as, for example, the group-by operator contains 4 to 9 sup-operators depending on the use-case. To evaluate different placements, we look at single operators and compare the best single-CU execution to the best heterogeneous placement of the corresponding sub-operators. To determine the heterogeneous placements, we do a full search of all possible placements for these sub-operators. We found, that in most cases the runtime is equal, i.e., sub-operators also choose a single-CU placement, while in a few cases we achieved speedups of up to 1.47x with the heterogeneous placement.

Regional optimization using execution islands: To evaluate the regional optimization, we use the runtime information of sub-operators to calculate the placement for the local, global, and regional approaches. As the global approach is not realistic considering varying cardinality estimations, we evaluate global optimization with certain errors in the cardinality. We multiply the real intermediate cardinalities either with 0.1x or with 10x to test the robustness of the optimization and estimation. Figure 9 shows the results including statistics on operators, sub-operators, and execution islands. We can see that global optimization is always better than local optimization, while global optimization with cardinality errors is slower than the original global optimization, caused by wrong runtime and transfer estimations. While global 0.1x still finds good placements, global 10x shows mostly bad performance. There, data is thought to be 10x larger leading to CPU heavy computation, because data does not need to be transfered for CPU execution. Local and regional optimizations are not affected by cardinality estimation errors, since cardinalities are known precisely, whereas regional optimization shows a performance similar to global optimization without errors.

Intermediate results with multiple locations: To evaluate the impact of allowing data copies, we use the SSB queries and test 1M random placements per query with and without allowing data copies. In the worst case, we see no speedup because of single-CU queries or placements, where by chance none of the copies could be exploited. In the best case, we achieve a speedup of 1.67x for highly heterogeneous queries. The performance gain of keeping copies depends on a query's transfer costs, as transfers are the only part that could be reduced. Single-CU execution has only transfers from the host, which can not benefit from data copies. Queries where the execution is significantly larger than the transfer costs do also not benefit much from data copies. As performance is not reduced in any case, we should allow copies in case a query is heavily dominated by transfers that could be avoided.

6.3 HERO Overheads

To evaluate the overhead of HERO, we compare the SSB runtimes of gpuDB with a HERO and a non-HERO execution. Since the original version of gpuDB supports only single-CU execution, we examine this case. Figure 10 shows the overhead in percentage for all combinations of SSB queries and CUs. Surprisingly, for some combinations the overhead is negative, meaning that the single-CU execution

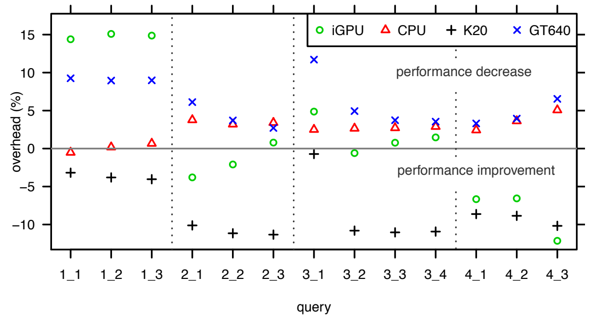


Figure 10: Relative overhead per query on a single CU ($\frac{totalHERO - totalNoHERO}{totalNoHERO}$). For better illustration, the points are grouped by vertical lines for similar queries (SSB groups 1-4). The overall average overhead is 0.5%.

is faster when using HERO, while others experience an overhead of up to 15% of the query runtime without HERO. There are two reasons for these variations:

(1) Queuing: HERO queues submitted sub-operators together, while, without HERO, OpenCL would start working on the sub-operators immediately when being submitted. The advantage of queuing and executing together is an aligned heterogeneous placement that is usually worth more than the small delay. However, for single-CU execution, the placement is fixed to a single CU and can not be improved by optimizing sub-operators together, making the delay visible as overhead. This delay depends on the query topology and on how eager the CUs start to work after submitting. For example in Figure 10, small queries show this overhead particularly (*1_1, 1_2, 1_3*), while iGPU and GT640 seem to suffer most from the queuing, showing that they usually execute work more eagerly than, e.g., the K20.

(2) Controlled Execution: When HERO is executing sub-operators, it is tightly controlling the execution on the CU. When a sub-operator is given to a CU, HERO is forcing the CU to execute it immediately without delay to ensure the correct time measurements for our runtime estimation. This can have two effects. An already eagerly executing CU might experience additional overheads through the active waiting for the result, leading to slower performance as without HERO (e.g., iGPU). On the other side, other CUs might experience a speedup through our approach if they are more lazy in execution (e.g., K20).

6.4 Performance and Placement Quality

For performance evaluation, we execute all SSB queries using HERO first on the CPU, before adding the K20, the GT640, and finally the iGPU. This way, we add more heterogeneity and allow HERO to distribute sub-operators to more CUs to improve the runtime. Figure 11 shows the performance results and the resulting placement distribution. As expected from the performance benchmark in Figure 8, the CPU is slow in execution. Adding the K20 improves the result of all 13 queries significantly, by placing most sub-operators on the GPU (speedup: avg 16.4x, max 24.9x). Adding the GT640 further improves the results for 11 queries, since the GT640 has a 10x faster connection to the main memory compared to the K20 (speedup to previous setting: avg 1.7x, max 2.3x). Finally, adding the iGPU can accelerate 10 queries additionally (speedup

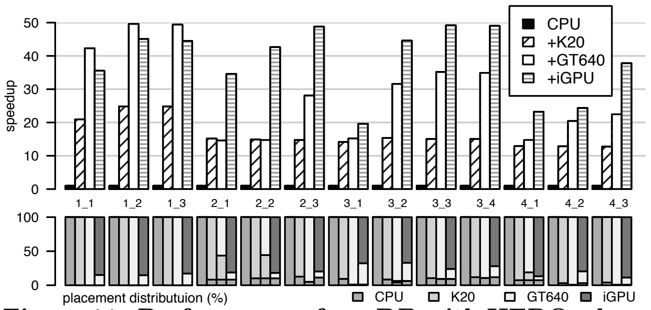


Figure 11: Performance of gpuDB with HERO when adding CUs for heterogeneous query execution.

to previous setting: avg 1.5x, max 2.9x). With the placement distribution, we see that some queries choose mainly single-CU placements, only differing in the chosen CU (*1_1, 1_2, 1_3, 4_2, 4_3*), while other queries choose more heterogeneous placements (e.g., *2_1 and 2_2*). Even for single-CU execution, an accurate placement optimization is needed to determine the best CU, as pure benchmark numbers are not descriptive enough. For example, the K20 is the most powerful CU in our system. However, if another GPU is available, the K20 is nearly never used because of the low transfer bandwidth to the system. All in all, adding placement optimization to the database system can achieve high speedups for the execution, for example, 50x for *Query 1_2*.

Limitation: In four out of the 13 queries, we see a small performance reduction when adding computing units (*Queries 1_1, 1_2, 1_3,* and *2_1*). There, we found that the overall placements are not ideal, while the regional placements for the sub-operators of each island are good. The reduction is caused by the adaptive placement approach using separate execution islands. The decisions for later islands depend on the decisions made for earlier islands, because they define the location of intermediate data. This is the same effect as presented in Section 4.2.2. For *Query 1_1*, the iGPU is mostly used throughout the query. However, most sub-operators of the first island run better on the GT640, so they are placed on this GPU. This introduces either being bound to the GT640 if transfers are too expensive or transferring the data to the iGPU, which would not be necessary if the first iteration would be executed on the iGPU in the first place. A global optimization could most likely avoid these problems, but would come with other drawbacks as stated earlier.

6.5 Adaptivity of Heterogeneous Placement

Besides the pure performance, we want to show the real benefits of our adaptive placement approach for heterogeneous execution using HERO. We exemplarily take *SSB Query 3_4* [25], which accesses four tables and includes three selections, three joins, a group by, and an order by. The selections are highly selective and the joins produce only small results, before the grouping reduces the result to two tuples. We now update the base data to produce more tuples in the selections, resulting in larger join results, which, at the end, are reduced by the grouping again to two tuples. Through this method, only the intermediate cardinalities change while the base table size and the final result size are constant. We compare the execution performance of a fixed placement, as a global optimizer would choose, and our adaptive approach, adjusting the placement according to the

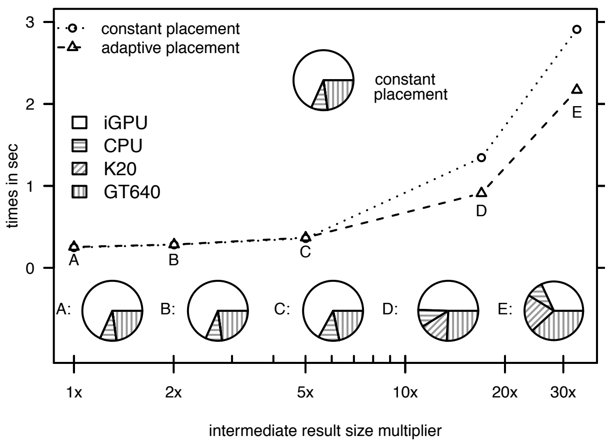


Figure 12: Adaptivity evaluation by changing the intermediate cardinalities of *SSB query 3_4*.

intermediately collected cardinalities. Figure 12 shows the actual runtime results and the placement distributions. The constant placement uses the cardinality information of the 1x scale. While the performance of the constant and adaptive placement is similar for small changes in intermediate cardinalities, larger changes show a significant difference. The pie charts show the placement distribution for each testing point. As expected, first the placements are similar to the constant placement, i.e., the majority of the sub-operators is on the iGPU, the placement changes towards more sub-operators on the K20 and the GT640 when the intermediate cardinalities grow. This adaption is not possible with global optimization and clearly shows the benefit of the adaptive approach.

6.6 Portability

Our implementation as virtualizing OpenCL driver enables a broad evaluation. Thus, we evaluated our approach with Ocelot as second DBMS as well. Ocelot [14] is an OpenCL extension to MonetDB [3] and is able to process nearly arbitrary queries on different CUs. It does that by altering the MonetDB query plan with MAL extensions, which makes MonetDB to hand off certain operators to Ocelot for external computation. Ocelot then uses OpenCL to execute the operator on a given CU. Like gpuDB, a query can only run on a single CU and Ocelot is currently also limited to one query at a time. For our performance evaluation, we use 9 different TPC-H queries with scale factor of 5.

Figure 13 shows the results. Compared to gpuDB, the CPU performs better, which is caused by mostly having two kernel variants with different memory-access patterns for CPUs and GPUs. Therefore, the kernels are more optimized for the used CU, as HERO switches these variants depending on the placement. For all queries, adding the K20 improves the result significantly (speedup avg: 4.0x, max: 14.8x), except for *Query 5*, where the CPU has already a comparable performance to the heterogeneous execution. Adding the GT640 improves the *Queries 3, 10, and 15*, while the others show that the combination of CPU and K20 is already ideal for the given workload and data sizes. Adding the iGPU improves only the result of *Query 3*. We can see in these results that HERO either improves the runtime with more heterogeneity or holds the performance, even

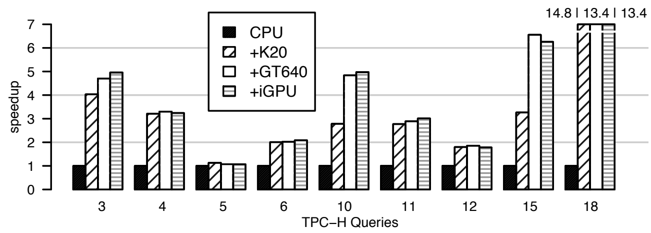


Figure 13: Performance of Ocelot with HERO when adding CUs for heterogeneous query execution.

if additional CUs can not be used beneficially for the given query. Only *Query 18* suffers from the explained limitation and the missing global view.

6.7 Summary

We have shown the efficiency of our adaptive placement using micro-benchmarks and full query execution. We presented that the HERO implementation can have a small overhead for single-CU execution depending on query topology and used CUs. However, for heterogeneous execution, the overhead is not visible since the query execution benefits from the heterogeneity, making the query faster. The performance tests with gpuDB and Ocelot have shown the benefits of our approach with speedups of up to 50x. Additionally, we have demonstrated the benefits of adjusting the placement according to changing intermediate cardinalities, resulting in reliably good performance.

7. RELATED WORK

Data Placement: Data placement approaches have been widely studied in the context of NUMA systems and distributed database systems. In NUMA systems, data and threads are placed together to avoid memory accesses beyond nodes and to spread the workload evenly between the nodes [2]. The main questions for placement decisions are "where is the data?" and "how busy are the nodes?", while the nodes are homogeneous in their computation but possibly heterogeneous in their interconnections [21]. For distributed systems, nodes and connections can be heterogeneous, while optimizations are mostly focusing on data placement according to the workload. Afterwards, the actual execution is done on the nodes holding the data [7].

Heterogeneous Placement: CoGaDB [5] and gpuQP [11] are prototypical database systems, optimizing the execution for hardware heterogeneity. CoGaDB uses a learning-based estimation model on operator level. gpuQP works with a small set of primitives, which are used to build larger operators, but uses benchmarking before execution to allow the estimation of individual runtimes. Both systems, CoGaDB and gpuQP optimize the heterogeneous execution globally in the query optimizer being dependent on estimated cardinality information.

Adaptive Query Processing: Our adaptive island-based approach has similarities with adaptive query processing [9], where the logical and physical plan can be re-optimized during run-time, if cardinality estimations differ too much from values experienced during execution. For our approach the QEP is fixed except the placement, which is reoptimized at run-time.

Implementation Approach: VirtCL [27] also implements an OpenCL driver to optimize the hardware underneath, however, it decides the placement locally by scheduling the next kernel on an available CU, where it might finish first. For database applications, where results of operators are reused by other operators, pure scheduling without further optimization is not beneficial. Wang et al. [26] propose a CUDA driver to schedule and load-balance multiple database operators of multiple queries. However, the CUDA approach is limited to NVIDIA GPUs, ignoring other CUs and the challenges of heterogeneous environments.

8. CONCLUSION

In this paper, we proposed adaptive work placement based on an execution island approach and two additional enhancements, improving runtime estimation through fine-grained learning and improving placement optimization through keeping temporary data copies. We combine our approaches in an adaptive placement sequence and show the effectiveness with an extended example. We implemented our techniques as a virtualization layer called HERO, which can be used by any OpenCL-based database system. In the evaluation, we have shown that our approach improves the performance by choosing good heterogeneous placements, while being adaptive to different intermediate cardinalities.

For **future work**, we strive to improve placement decisions near island borders and overcome the limitations shown in the evaluation, by allowing our adaptive optimizer to look beyond its island and potentially choose a more optimal placement. This could be possible by defining an initial placement using inaccurate cardinality estimations of the database optimizer and propose placements for all sub-operators, while afterwards applying our island approach to improve these placement decisions. However, errors in the cardinality information will have a significant influence on the initial placement decisions. Depending on the magnitude of the error, the initial placement can also negatively influence our adaptive placement decisions. Besides improving the placement quality, our approach can be implemented into the database optimizer, where it is possible to couple adaptive placement optimization with physical operator adjustments and query restructuring known from adaptive query processing [9]. Additionally, all collected information like correct cardinalities can be fed back into the database optimizer to improve later cardinality estimation.

Acknowledgments. This work is funded by the German Research Foundation (DFG) within the Cluster of Excellence "Center for Advancing Electronics Dresden" (Orchestration Path). Parts of the evaluation hardware were generously provided by the Dresden GPU Center of Excellence. We also thank the anonymous reviewers for their constructive feedback to improve this paper.

9. REFERENCES

[1] D. Abadi, P. Boncz, S. Harizopoulos, S. Idreos, and S. Madden. The Design and Implementation of Modern Column-Oriented Database Systems. In *Foundations and Trends in Databases*, volume 5, pages 197–280, 2013.

[2] J. Antony, P. P. Janes, and A. P. Rendell. Exploring Thread and Memory Placement on NUMA Architectures: Solaris and Linux, UltraSPARC/FirePlane and Opteron/Hypertransport. In *Proceedings of HiPC*, pages 338–352, 2006.

[3] P. A. Boncz, M. L. Kersten, and S. Manegold. Breaking the Memory Wall in MonetDB. *Communications ACM*, 51(12):77–85, Dec. 2008.

[4] P. A. Boncz, M. Zukowski, and N. Nes. MonetDB/X100: Hyper-Pipelining Query Execution. In *CIDR:225-237*, 2005.

[5] S. Breß. The Design and Implementation of CoGaDB: A Column-oriented GPU-accelerated DBMS. *Datenbank-Spektrum*, 14(3):199–209, 2014.

[6] S. Breß and G. Saake. Why It is Time for a HyPE: A Hybrid Query Processing Engine for Efficient GPU Coprocessing in DBMS. *Proc. VLDB Endow.*, 6(12):1398–1403, Aug. 2013.

[7] A. Brinkmann, K. Salzwedel, and C. Scheideler. Compact, Adaptive Placement Schemes for Non-uniform Requirements. In *Proceedings of SPAA*, pages 53–62. ACM, 2002.

[8] S. Christodoulakis. Implications of Certain Assumptions in Database Performance Evaluation. *ACM Trans. Database Syst.*, 9(2):163–186, June 1984.

[9] A. Deshpande, Z. Ives, and V. Raman. Adaptive Query Processing. *Found. Trends databases*, 1(1):1–140, Jan. 2007.

[10] H. Esmaeilzadeh, E. Blem, R. St. Amant, K. Sankaralingam, and D. Burger. Dark silicon and the end of multicore scaling. In *Proceedings of ISCA*, pages 365–376. ACM, 2011.

[11] B. He, M. Lu, K. Yang, R. Fang, N. K. Govindaraju, Q. Luo, and P. V. Sander. Relational Query Coprocessing on Graphics Processors. *ACM Trans. Database Syst.*, 34(4):21:1–21:39, Dec. 2009.

[12] B. He, K. Yang, R. Fang, M. Lu, N. Govindaraju, Q. Luo, and P. Sander. Relational Joins on Graphics Processors. In *Proceedings of the 2008 ACM SIGMOD*, SIGMOD '08, pages 511–524, New York, NY, USA, 2008. ACM.

[13] J. He, S. Zhang, and B. He. In-cache Query Co-processing on Coupled CPU-GPU Architectures. *Proc. VLDB Endow.*, 8(4):329–340, Dec. 2014.

[14] M. Heimel, M. Saecker, H. Pirk, S. Manegold, and V. Markl. Hardware-oblivious Parallelism for In-memory Column-stores. *Proc. VLDB Endow.*, 6(9):709–720, July 2013.

[15] Y. E. Ioannidis and S. Christodoulakis. On the propagation of errors in the size of join results. In *Proceedings of ACM SIGMOD*, pages 268–277. ACM, 1991.

[16] S. Jha, B. He, M. Lu, X. Cheng, and H. P. Huynh. Improving Main Memory Hash Joins on Intel Xeon Phi Processors: An Experimental Approach. *Proc. VLDB Endow.:642–653*, 2015.

[17] T. Karnagel, D. Habich, and W. Lehner. Local vs. Global Optimization: Operator Placement Strategies in Heterogeneous Environments. In *Proceedings of the Workshops of the EDBT/ICDT*, pages 48–55, 2015.

[18] T. Karnagel, D. Habich, B. Schlegel, and W. Lehner. Heterogeneity-Aware Operator Placement in Column-Store DBMS. *Datenbank-Spektrum*, 14(3):211–221, 2014.

[19] V. Leis, P. Boncz, A. Kemper, and T. Neumann. Morsel-driven parallelism: a NUMA-aware query evaluation framework for the many-core age. In *Proceedings of the 2014 ACM SIGMOD*, pages 743–754. ACM, 2014.

[20] V. Leis, A. Gubichev, A. Mirchev, P. Boncz, A. Kemper, and T. Neumann. How Good Are Query Optimizers, Really? *Proc. VLDB Endow.*, 9(3):204–215, Nov. 2015.

[21] B. Lepers, V. Quéma, and A. Fedorova. Thread and Memory Placement on NUMA Systems: Asymmetry Matters. In *Proceedings of the 2015 USENIX*, pages 277–289, 2015.

[22] S. Meraji, B. Schiefer, L. Pham, L. Chu, P. Kokosielis, A. Storm, W. Young, C. Ge, G. Ng, and K. Kanagaratnam. Towards a Hybrid Design for Fast Query Processing in DB2 with BLU Acceleration Using Graphical Processing Units: A Technology Demonstration. In *Proceedings of SIGMOD*, pages 1951–1960. ACM, 2016.

[23] R. Mueller, J. Teubner, and G. Alonso. Data Processing on FPGAs. *Proc. VLDB Endow.*, 2(1):910–921, Aug. 2009.

[24] T. Neumann. Efficiently compiling efficient query plans for modern hardware. *Proc. VLDB Endow.*, 4(9):539–550, 2011.

[25] P. ONeil, E. ONeil, X. Chen, and S. Revilak. The star schema benchmark and augmented fact table indexing. In *Technology Conference on Performance Evaluation and Benchmarking*, pages 237–252. Springer, 2009.

[26] K. Wang, K. Zhang, Y. Yuan, S. Ma, R. Lee, X. Ding, and X. Zhang. Concurrent Analytical Query Processing with GPUs. *Proc. VLDB Endow.*, 7(11):1011–1022, July 2014.

[27] Y.-P. You, H.-J. Wu, Y.-N. Tsai, and Y.-T. Chao. VirtCL: A Framework for OpenCL Device Abstraction and Management. In *Proceedings of PPoPP*, pages 161–172. ACM, 2015.

[28] Y. Yuan, R. Lee, and X. Zhang. The Yin and Yang of Processing Data Warehousing Queries on GPU Devices. *Proc. VLDB Endow.*, 6(10):817–828, Aug. 2013.