Control and Cybernetics

vol. **41** (2012) No. 4

A framework for cost based optimization of hybrid CPU/GPU query plans in database systems[*][†]

by

Sebastian Breß, Ingolf Geist, Eike Schallehn, Maik Mory and Gunter Saake

Otto-von-Guericke University Magdeburg, Universitätsplatz 2, D-39106 Magdeburg
{bress,geist,eike,maik.mory,saake}@iti.cs.uni-magdeburg.de

Abstract: Current database research identified the use of computational power of GPUs as a way to increase the performance of database systems. As GPU algorithms are not necessarily faster than their CPU counterparts, it is important to use the GPU only if it is beneficial for query processing. In a general database context, only few research projects address hybrid query processing, i.e., using a mix of CPU- and GPU-based processing to achieve optimal performance. In this paper, we extend our CPU/GPU scheduling framework to support hybrid query processing in database systems. We point out fundamental problems and propose an algorithm to create a hybrid query plan for a query using our scheduling framework. Additionally, we provide cost metrics, accounting for the possible overlapping of data transfers and computation on the GPU. Furthermore, we present algorithms to create hybrid query plans for query sequences and query trees.

1. Introduction

Graphics Processing Units (GPUs) are specialized processors designed to support graphical applications. GPUs have advanced capabilities of parallel processing and have more computing power than CPUs nowadays. Using GPUs to speed up generic applications is called General Purpose Computation on Graphics Processing Units (GPGPU). In particular, parallelizable applications benefit from computations on the GPU (Sanders & Kandrot, 2010).

Current research focuses on the acceleration of database systems by using the GPU as co-processor (Bakkam & Skadron, 2010; He et al., 2008, 2009; Pirk, 2012; Pirk, Manegold & Kersten, 2011; Walkowiak et al., 2010) GPUs are utilized for accelerating query processing like relational operations (Bakkam & Skadron, 2010; Diamos et al., 2012; Govindaraju et al., 2006; He et al., 2006, 2008; He & Yu, 2011; Kaldewey et al., 2012; Pirk, 2012; Pirk, Manegold & Kersten, 2011; Pirk et al., 2012), XML path filtering (Moussalli et al., 2011), online aggregation (Lauer et al., 2010), compression

[*]This paper is an extended version of previous work, Breß, Schallehn & Geist (2012).

[†]Submitted: October 2012; Accepted: November 2012.

(Andrzejewski & Wrembel, 2010; Fang, He & Lao, 2010) and scans (Beier, Kilias & Sattler, 2012), as well as query optimization, e.g., GPU based selectivity estimation (Augustyn & Zederowski, 2012; Heimel & Markl, 2012).

However, the data transfer between CPU and GPU memory introduces a large overhead leading to a better performance of CPU algorithms for relatively small data sets (Gregg & Hazelwood, 2011). Therefore, typical plans for a database query consists of a combination of GPU and CPU algorithms. We call such a query plan a *hybrid query*.

We have to solve many problems to find a hybrid query plan that allows for an efficient usage of the GPU as co-processor during database query processing. Therefore, we need a hybrid query optimizer (Heimel, 2011) to construct a good hybrid query plan. The optimizer uses a cost model, which includes GPU and CPU costs. Scheduling operations to GPU or CPU increases the search space for an optimizer. Hence, we have to reduce the search space by using two-step approaches or other heuristics.

In previous work, we presented a self-tuning decision model, which distributes database operations response time minimal among CPU and GPU processing units (Breß, Schallehn & Geist, 2012). The model is a black box approach that computes estimated execution times for algorithms using statistical methods and observed execution times. So far, we only considered single operations. In this paper, we will present an extension how a hybrid query plan with low response time is constructed from a logical query plan using the scheduling framework.

This paper is an extended version of prior work (Breß, Schallehn & Geist, 2012). It summarizes our decision model (Breßet al., 2012; Breß, Mohammad & Schallehn, 2012) and our cost estimation approach for hybrid query plans for effective GPU coprocessing in relational DBMS (Breß, Schallehn & Geist, 2012). Current GPUs support concurrent processing and data transfer, which can reduce the overall execution time (AMD, 2011; NVIDIA, 2012). Therefore, we contribute a new cost metric for the computation of query response time assuming concurrent processing of database operations and data transfer on the GPU are possible. Furthermore, we extend our optimization algorithms from query sequences (query plan as a sequence of operations) to query trees (query plan as operator tree).

The remainder of the paper is structured as follows. First, we present the necessary background in Section 2.1. In Section 2.2, we discuss basic problems that occur during the processing and optimization of hybrid queries. We introduce our notation in Section 3. We give a short overview of the decision model in Section 4 and present an approach for the construction of sequential hybrid query plans in Section 5. Afterwards, we present our extended cost metrics and algorithms that consider possible concurrency of data transfer and computation on GPU side in Section 6 and utilize them in a new heuristic, which we describe in Section 7. In Section 8, we generalize our concepts from query sequences to query trees. The paper closes with a discussion of related work in Section 9, a discussion of future research steps in Section 10, and a conclusion in Section 11.

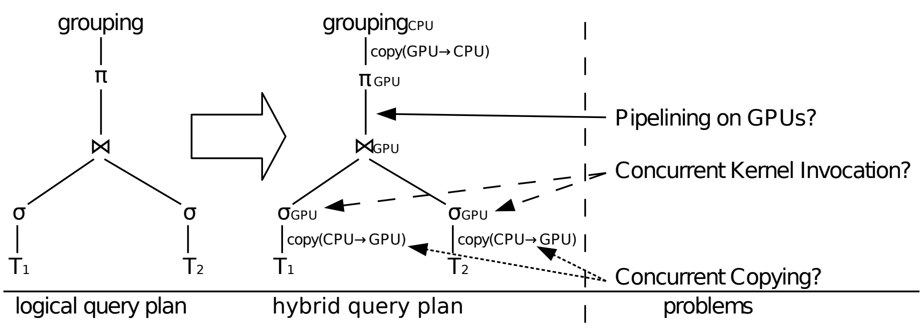


Figure 1. Example: hybrid query plan and problems of hybrid query processing

2. Preliminaries

In this section, we provide a brief overview over graphics processing units and challenges for hybrid query processing.

2.1. Graphics Processing Units

Graphics Processing Units (GPUs) are specialized processors designed to support graphical applications. In contrast to the CPU, the GPU is optimized for throughput, which is achieved by massively parallel execution using large numbers of threads. Furthermore, the GPU is optimized for numerical computation, but control flow statements brake the performance of a GPU algorithm. Hence, not all applications benefit from GPU (Sanders & Kandrot, 2010).

A GPU can only process data that resides in the GPU memory. Hence, data has to be transferred from CPU main memory to the GPU memory before processing on the GPU. After the GPU processed the data, the result has to be transferred back into the CPU memory (NVIDIA, 2012). The copy operations introduce an overhead, which can lead to a higher total execution time of a GPU algorithm compared to its CPU counterpart, even if the execution on the GPU is faster than on the CPU (Gregg & Hazelwood, 2011).

2.2. Challenges for hybrid query processing

The main problem of hybrid query processing is to use the GPU only if it is beneficial for the performance of a query. The physical optimization process in database query processing should be revised to enable an effective usage of the GPU to increase the performance of database systems. It is difficult to generalize query processing from pure CPU based processing to a hybrid CPU/GPU solution. One possible approach estimates the execution times of all algorithms for an operation, choosing for each operation in a query the algorithm with the lowest expected costs.

If a GPU algorithm is selected, then additional communication costs will be incurred depending on the data storage location (Gregg & Hazelwood, 2011). We discuss two common approaches.

First, is cost based optimization by pruning the optimization space and comparing the costs of candidate query plans. Therefore, we need to create a set of hybrid query plan candidates and then choose the plan with lowest costs. To keep the overhead low, we have to reduce the optimization space while keeping promising candidates. Hence, we need a cost model that can compute the cost of a hybrid query plan in consideration of data storage location and possibly parallel data transfer and data processing. We discuss our cost metrics in Section 6 and discuss a cost based optimization algorithm for query sequences in Section 7.

Second is a greedy strategy which computes exactly one hybrid query plan. Considering the growth of the optimization space, the overhead of a cost based approach is likely to be high. Hence, we present a greedy approach in Section 5 for query sequences and in Section 8 for query trees.

The greedy strategy introduces lower overhead, whereas the cost based approach is likely to find a query plan with lower cost. Fig. 1 illustrates how a hybrid query plan is created from a logical query plan. Note the necessary copy operations, if the optimizer decides to change the processing device (CPU/GPU). We identify five problems:

Pipelining challenge Modern GPUs can enqueue kernels and concurrently process them, but the inter-kernel communication is undefined (NVIDIA, 2012). Hence, a regular pipelining between two GPU algorithms is not possible. However, it is possible to integrate two operations into one kernel. In this case, several kernels are combined and compiled together at run-time, if OpenCL is used (Heimel, 2011).

Execution time prediction challenge Database operations can be executed in parallel, e.g., in Fig. 1, where two selections can be processed concurrently. The concurrent processing of kernels is possible for current GPUs (NVIDIA, 2012), but it is hard to predict the influence on execution times.

Copy serialization challenge Concurrent copy operations in the same direction are not allowed (NVIDIA, 2012). As Fig. 1 illustrates, concurrent data transfer occurs in query plans. Hence, copy operations have to be serialized, and the following selections have to be serialized as well. A possible approach is to combine the two data streams in one copy operation and reorganize the data in the GPU memory. In this way, the PCIe Bus is better utilized.

Critical query challenge Since the number of concurrent kernel executions (16 by current NVIDIA GPUs, NVIDIA, 2012) and the PCIe Bus bandwidth are limited, not every query benefits from the GPU. Thus, a heuristic is needed, which chooses ”critical queries” that, first, benefit from the GPU usage and, second, have a certain degree of ”importance”, because some queries require higher performance than others.

Optimization impact challenge A further problem is the estimation how the execution of one query influences the performance of another hybrid query. We do not consider this problem here and address it in future work.

3. Notation

Let O be a database operation and let $AP_O = \{A_1,..,A_m\}$ be an algorithm pool for operation O, where each algorithm A_i in the algorithm pool is executable either on the CPU or the GPU. The model assumes that the performance of an algorithm depends on the input data set D, which abstracts the features of the real data set. This means a data set contains all statistical information of the represented real data set. Examples are the datasize, the data distribution, or the selectivity, if a selection operation is performed on the data. Note that for the selection, the selectivity has to be estimated according to the operations parameters. Examples of selectivity estimation can be found in Augustyn & Zedrowski (2012), Getoor, Taskar & Koller (2001), Heimel & Markl (2012). Let $T_{est}(A,D)$ be the estimated and $T_{real}(A,D)$ be the measured execution time of algorithm A for a data set D. Let MPL_A be a measurement pair list containing all current measurement pairs $(D,T_{real}(A,D))$ of algorithm A.

A data set D is partitioned in n parts P_i, where the parts are disjoint ($P_i \cap P_j = \emptyset$ with $i \neq j$) and complete ($D = P_1 \cup P_2 \cup \cdots \cup P_n$). The parts have to be disjoint, because otherwise the same data has to be processed more than once. Overlapping of parts also leads to wrong results. The partitioning has to be complete, because otherwise it cannot be guaranteed that the complete data set D is processed. Note that the definition allows that a data set D be a part P of itself ($P = D \wedge P \subseteq D$). The times to copy a part P_i completely from the CPU main memory to the GPU memory or vice versa are denoted $T_{cpy}(P_i)$ and $T_{cpyb}(P_i)$, respectively. The estimated time an algorithm A needs to process a part P_i is $T_{comp}(P_i,A)$. The result of an operation O for an input part P_i is denoted as $P_{result,i} = O(P_i)$. Let $NG(P_i)$ (not in GPU RAM) be a function that returns 1 if and only if a part is not stored in the GPU RAM. Let $FR(P_{result,i})$ (final result) be a function that returns 1 if and only if the resulting part $P_{result,i}$ is a final result.

We now introduce query sequences and query trees. A logical query sequence $QS_{log} = O_1 O_2 \cdots O_n$ is a sequence of operations to be executed. Then, QS_{hybrid} is a hybrid sequence query, if each operation in QS_{log} is replaced with an algorithm A. Each algorithm uses either the CPU or the GPU. A logical query tree QT_{log} is the result of a logical query optimization using a traditional database optimizer. A hybrid query tree QT_{hybrid} is constructed from QT_{log} by assigning to each node in QT_{log} an algorithm. In the case of GPU algorithms, necessary copy operations are inserted in the query tree. Table 1 summarizes the notation.

4. Decision model

In this section, we provide a brief overview of our decision model, which we introduced in previous work (Breßet al., 2012; Breß, Mohammad & Schallehn, 2012).

4.1. Overview

Every year, new features are introduced in GPUs. Hence, it becomes increasingly complex to create analytical cost models to estimate the execution time of a GPU al-

Table 1. Notation used

Symbol	Description
D	Data set
A	Algorithm
O	Operation
AP_O	Algorithm pool for O
$T_{est}(A,D)$	Estimated execution time of A for D
$T_{real}(A,D)$	Measured execution time of A for D
P_i	Part i of data set D
$P_{result,i} = O(P_i)$	Result part
$NG(P_i)$	Function, returns true if and only if P_i is stored in GPU RAM
$FR(P_{result,i})$	Function, returns true if and only if $P_{result,i}$ is not processed by the GPU anymore
QS_{log}	Logical query sequence
QS_{hybrid}	Hybrid query sequence
QT_{log}	Logical query tree
QT_{hybrid}	Hybrid query tree

gorithm, which was done by, e.g., Baghsorkhi et al. (2010), He et al. (2009), Hong & Kim (2009), Kothapalli et al. (2009), Schaa & Kaeli (2009) and Zhang & Owens (2011).

Furthermore, a GPU algorithm is not necessarily faster than its CPU counterpart, mainly due to the overhead of data transfers (Gregg & Hazelwood, 2011). To decide on the algorithm with lowest execution time, we introduced a self-tuning decision model (Breßet al., 2012; Breß, Mohammad & Schallehn, 2012) and observed significant performance improvements depending on the workload. Since the decision model is a central component of our framework, we will provide a brief summary.

Our model uses a learning based approach, to counter the problem of increasing complexity for analytical cost models. The basic idea is to observe the execution behavior of algorithms and deduce estimated execution times from past measured execution times. Hence, an algorithm is the central component of abstraction. The model learns the characteristic execution time curve of an algorithm for a specific data set D.

Let O be an operation and $AP_O = \{A_1, .., A_m\}$ an algorithm pool, which contains all available algorithms to execute O. Note that each algorithm uses either the CPU or the GPU, but not both. He et al. (2009) discovered that no significant performance gain can be achieved by processing the same operation on both processing units by dividing the operation into two parts, where one part is processed on the CPU and the other on the GPU. By choosing an algorithm that uses a certain processing unit, the corresponding operation is processed by the CPU or the GPU.

Be $T_{est}(D,A)$ the estimated and $T_{real}(D,A)$ the measured execution time of an algorithm that processes a data set D. A measurement pair $MP = (D, T_{real}(D,A))$ is a tuple

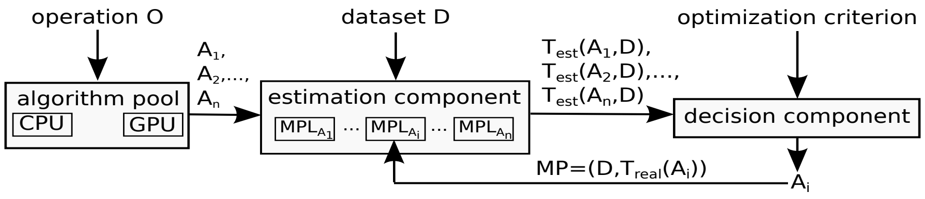


Figure 2. Overview of the decision model

of a data set D and $T_{real}(D,A)$ the measured execution time of an algorithm A.

4.2. Architecture

An incoming operation O is passed to an algorithm pool which passes all available algorithms to process O to an estimation component. The estimation component has the data set D that is to process as additional input parameter and derives estimated execution times for each available algorithm for the specified data set D. These estimated execution times are then passed to a decision component, which decides on the optimal algorithm by using a user specified optimization criterion OC. Note that the execution time $T_{real}(D,A_i)$ of the selected algorithm A_i is measured and is inserted in the measurement pair list of A_i together with the features of the data set D. The feedback loop enables our model to refine future estimations by collecting measurement pairs. Fig. 2 summarizes the architecture of our model.

4.3. Estimation component

To enable the estimation component to compute estimated execution times without using analytical cost models, we have to specify three parameters for each algorithm: (1) a statistical method, (2) an approximation function $F_A(D)$, which is dictated by the statistical method, and (3) a measurement pair list MPL_A, which contains recent observations of the algorithms execution. Our model updates the approximation function of an algorithm by applying the assigned statistical method to the measurement pair list of the algorithm. More details are available in Breß, Mohammad & Schallehn (2012).

4.4. Decision component

The decision component currently supports only response time as possible optimization criterion OC. Hence, our model tries to select the algorithm for execution that has the lowest execution time. We implemented the response time criterion by selecting the algorithm that is most likely to be the fastest. Therefore, we let the model choose the algorithm with the lowest estimated execution time to execute operation O.

5. Constructing hybrid query sequences

We present a greedy approach to construct a hybrid query sequence using our decision model. The approach does not guarantee optimal results, but introduces only a low overhead. We assume for simplicity that a logical query sequence is a sequence of operations $QS_{log} = O_1O_2 \cdots O_n$. We construct a hybrid query sequence by choosing for each operation O_i in QS_{log} the response time minimal algorithm, which leads to a hybrid query sequence QS_{hybrid}. Depending on whether an algorithm uses the CPU or GPU, the operation is executed on the corresponding processing unit. Let $CA(D,O)$ be a function, which chooses the fastest algorithm A for a given data set D and an operation O. It uses the function T_{est} to compute the estimated execution times for the algorithms. T_{est} considers the time needed to copy data to and from the GPU memory in the case a GPU algorithm is selected. Hence, $CA(D,O)$ chooses a GPU algorithm only if the execution time of a CPU algorithm is higher than the execution time of a GPU algorithm and the needed data transfer times together. Let $CAS(A)$ be a function that returns an algorithm sequence needed to execute algorithm A. In case of a CPU algorithm, $CAS(A)$ returns A. In the case of a GPU algorithm, $CAS(A)$ returns a sequence of three algorithms. The first is $A_{cpy}(D)$, which copies the input data from the CPU RAM to the GPU RAM (host to device). The second is $A_{i,GPU}(D)$ that processes the data set D on the GPU. The third is $A_{cpyb}(D_{result,i})$, which transfers the result set back to the CPU RAM (device to host). In case of a CPU algorithm, operation O_i is substituted by $A_{i,CPU}(D)$.

$$T_{est}(D,A) = \begin{cases} T_{est}(D,A) & \text{if } A = A_{CPU} \\ T_{est}(A_{cpy}(D)A(D)A_{cpyb}(D_{result})) & \text{otherwise} \end{cases} \tag{1}$$

$$CA(D,O) = A \text{ with } T_{est}(D,A) = min\{T_{est}(D,A) | A \in AP_O\} \tag{2}$$

$$CAS(A) = \begin{cases} A(D) & \text{if } A = A_{CPU} \\ A_{cpy}(D)A(D)A_{cpyb}(D_{result}) & \text{otherwise.} \end{cases} \tag{3}$$

We formalize our approach in Algorithm 1. In lines *1–6*, we construct the optimal query sequence using the functions $CA(D,O)$ and $CAS(A)$ of our decision model by choosing the best expected algorithm for each operation in the query. The algorithm leads to two succeeding copy operations in different directions, when two succeeding operations are executed on the GPU. This unnecessary copy operations are removed by the algorithm in lines 7–11.

Example: For the following example, we omit the data sets in the algorithm notation. We consider selections (S), projections (P), joins (J), and groupings (G). The query plan from Fig. 1 as query sequence is written like this: $O_SO_SO_JO_PO_G$. The following hybrid query sequence is the result of the first loop in algorithm:

$$A_{S,CPU}A_{S,CPU}A_{cpy}A_{J,GPU}A_{cpyb}A_{cpy}A_{P,GPU}A_{cpyb}A_{G,CPU}.$$

Algorithm 1 Construction of QS_{hybrid} from QS_{log} with the Greedy Algorithm

Input: $QS_{log} = (O_1, D^1); \cdots ; (O_n, D^n)$
Output: $QS_{hybrid} = A_1 \cdots A_m$

```
 1: $QS_{hybrid} = \emptyset$
 2: for $O_i$ in $QS_{log}$ do
 3:     $A = CA(D^i, O)$
 4:     $AS = CAS(A)$
 5:     append $AS$ to $QS_{hybrid}$
 6: end for
 7: for $A_i$ in $QS_{hybrid}$ do
 8:     if ($A_i = A_{cpyb}(D)$ and $A_{i+1} = A_{cpy}(D)$) then
 9:         delete $A_i A_{i+1}$ from $QS_{hybrid}$
10:     end if
11: end for
```

After the removal of unnecessary copy operations in the second loop of the algorithm, the final result is

$$A_{S,CPU} A_{S,CPU} A_{cpy} A_{J,GPU} A_{P,GPU} A_{cpyb} A_{G,CPU} .$$

Since the decision model decided to use a GPU algorithm in two cases, we can assume that the response time of the hybrid plan is smaller than the time of the pure CPU plan.

Discussion of the greedy algorithm: Our proposed algorithm is not guaranteed to generate an optimal hybrid query sequence in all cases for this problem. Executing a single operation on the GPU might be more expensive than using the CPU. However, executing a sequence of operations on the GPU may be faster than executing them entirely on the CPU. We consider for the cost computation no concurrent copying and processing and hence, sum up the times of all algorithms in a plan to compute the execution time of a query sequence. In this example, we will use the execution times shown in Table 2. Consider the query sequence $O_S O_S O_J O_P O_G$ and assume the algorithm processes O_J. Then $T_{est}(A_{cpy} A_{J,GPU} A_{cpy})$ is greater than $T_{est}(A_{J,CPU})$ ($3+2+3=8>5$) and the algorithm decides for the CPU algorithm for the Join. However, if the algorithm had considered O_J and the successor O_P, then it would have seen that $T_{est}(A_{cpy} A_{J,GPU} A_{P,GPU} A_{cpyb})$ is less than $T_{est}(A_{J,CPU} A_{P,CPU})$ ($3+2+1+3=9<5+5$), so the usage of the GPU algorithms for the Join and the Projection would result in a cheaper query sequence. Since the algorithm only chooses locally optimal solutions and does not look forward in the query sequence, it cannot consider the possibility that the selection of a slower algorithm could lead to a faster query sequence, because it cannot foresee the copy operation optimization. However, the algorithm is able to create a promising candidate, for evolutionary or randomized optimization algorithms.

Table 2. Example execution times of algorithms for the given example data sets

Processing unit	O_S	O_J	O_P	O_G	O_{cpy}	O_{cpyb}
CPU	1	5	5	2	3	3
GPU	3	2	1	7	-	-

6. Cost metric for computation of response time for query sequences

The use of concurrent GPU kernel execution and data transfer using page locked host memory (NVIDIA, 2012) mitigates the negative impact of expensive copy operations. In order to enable an optimizer to use this technique, cost metrics for computation of total and response times of a query have to be developed. For this, we extend our concept using sequential data transfer and GPU computation to parallel data transfer and GPU computation. To the best of our knowledge, concurrent kernel execution and data transfer of the GPU are not considered in cost metrics in prior work. Ilić et al. (2011) report that they take into account the overlapping of computation and communication. The authors claim that the performance approximations can accurately model the real and improved performance of the GPU. However, they did not describe their metrics. Hence, we provide the necessary metrics in this paper.

The input of the cost formulas are the estimated execution times of algorithms for a given data set and device. The estimation component of our decision model provides these times. To learn and improve the estimations, the real execution times of every algorithm in a query plan are collected and added as measurements pairs to the estimation component.

6.1. Extension of existing metrics

We now extend the sequential metrics. In general, a GPU can only process a data set, if it is completely stored in the GPU RAM. However, in a database context we can mitigate this restriction by partitioning the data set. That allows parallel copying and processing of different parts. Partioning is possible for operations like selection, projection, and aggregation. As the GPU RAM is comparably small compared to the CPU RAM, it is beneficial to concurrently transfer data to the GPU, process the data on the GPU, and copy processed data back to the CPU RAM.

Traditional approaches (He et al., 2009; Kothapalli et al., 2009; Schaa & Kaeli, 2009) model the cost of a database operation O by using the GPU algorithm A_{GPU} as follows. The execution time of a GPU algorithm is the sum of the time needed to copy the input data from the CPU RAM to the GPU RAM ($T_{cpy}(D)$), the time to process the data set D ($T_{comp}(D,A)$), and the time needed to transfer the result data from the GPU RAM back to the CPU RAM ($T_{cpyb}(D_{result})$),

$$T_{est}(D,A) = T_{cpy}(D) + T_{comp}(D,A) + T_{cpyb}(D_{result}). \quad (4)$$

Equation (4) does not consider the capability of GPUs to concurrently transfer data between CPU and GPU RAM and to process data on the GPU. Since the relative time for copying data compared to the execution time of the GPU kernel increases with processing power of the GPU (Gregg & Hazelwoód, 2011), such a fixed cost metric could lead to the decision to use the CPU, while the GPU would have been faster if the concurrent data copying and processing would have been considered. Hence, this metric is not suitable for the cost computation of a single operation and a query, respectively.

For the new cost metric, we assume that the data is partitioned or can be quickly partitioned. Furthermore, the new metric distinguishes between final results (result of a query) and intermediate results. If data is processed on the GPU and the result is an intermediate data set, then this data can be processed by the next operation on the GPU without the necessity to transfer data from the CPU RAM to the GPU RAM. If some data is still missing, e.g., the second table needed for a join, then this data has to be copied from the CPU to the GPU RAM.

For example, consider the selection on a table T_1 that is followed by a join with a second table T_2, denoted as $J(T_1,T_2)$. If the selection is performed on the GPU, then the join $J(T_1,T_2)$ can be processed without any additional copying cost if T_1 and T_2 are located in the GPU RAM. If T_2 is not yet in the GPU RAM, it has to be copied from the CPU RAM. If the data is partitioned, the GPU can start the join processing after the first part of T_2 arrived in the GPU RAM. This principle was used by Pirk, Manegold& Kersten (2011), too. However, caching of intermediate results would only be possible, if there is enough space available in the GPU RAM. For large data sets, it is possible that an execution of an operation needs the whole GPU RAM, or even a partitioning of the input data becomes necessary, if the whole data set does not fit in the GPU RAM. Therefore, the physical constraints of the hardware have to be considered during optimization process. We now extend the traditional metrics for total and response time computation considering partitioning and concurrent data transfer.

6.2. Computation of total execution time

For total execution time computation, we extend the metric in Equation (4) to the partitioning approach. We do not consider partitioning time, because we assume it is negligible. A data set D is partitioned into n parts $P_1, P_2, \ldots, P_n$. This results in Equation (5) for total execution time of a GPU algorithm:

$$\begin{aligned} T_{total}(D,A) = & \left(\sum_{i=1}^{n} T_{cpy}(P_i) \cdot NG(P_i) \right) + \sum_{i=1}^{n} T_{comp}(P_i, A) \\ & + \left(\sum_{i=1}^{n} T_{cpyb}(O(P_i)) \cdot FR(O(P_i)) \right). \end{aligned} \tag{5}$$

The total execution T_{total} time consists of the sum of the execution times of each part. Thereby, we consider the location of a part. If a part P_i is located in the GPU

RAM, the transfer time $T_{cpy}(P_i) \cdot NG(P_i)$ is zero ($NG(P_i) = 0$). Equally, if the result is not final and reused in a later operation, the data will not be copied back. So, $T_{cpyb}(O(P_i)) \cdot FR(O(P_i))$ is zero in this case.

6.3. Computation of response time for single operations

Equation (5) does not consider the concurrent execution of data transfer to and from GPU RAM and processing on the GPU. The steps that can be done concurrently are not considered in (5). Let the data set D be partitioned into $P_1, P_2, \ldots, P_n$. The algorithm A processes D on the GPU. The GPU algorithm starts the processing of D directly after the first part P_1 has been completely transferred into the GPU RAM. The corresponding result $P_{result,1}$ is either transferred back to the CPU RAM or kept in the GPU RAM if it will be needed in a subsequent operation. After this initialization step, the execution time of subsequent processing parts P_{i+1} is the maximum time of the data transfer of the following part P_{i+2} to GPU RAM, the computation of the part P_{i+1}, or transfer back of the last part P_i to the CPU RAM. We summarize this in the function

$$\max(\max(T_{cpy}(P_{i+2}), T_{comp}(P_{i+1}, A), T_{cpyb}(P_{result,i})).$$

Besides the initialization step, we also have to process serially the last part, i.e., the GPU processing of part P_n and the data transfer of $P_{result,n}$. Furthermore, we will include the location of a part into the basic formula by using the function $NG(P_i)$ and $FR(P_{result,i})$. If a part is already in the GPU RAM ($NG = 0$), we do not have to transfer it. If a result is not a final result ($FR(P_{result,i}) = 0$), we keep the data in the GPU RAM.

Equation (6) summarizes all concepts and provides the computation of the response time of an algorithm A for a partitioned data set $D = P_1 P_2 \cdots P_n$.

$$\begin{aligned} T_{resp}(D,A) = {} & T_{cpy}(P_1) \cdot NG(P_i) + \max(T_{cpy}(P_2) \cdot NG(P_i), T_{comp}(P_1, A)) \\ & + \sum_{i=1}^{n-2} \max(T_{cpy}(P_{i+2}) \cdot NG(P_i), T_{comp}(P_{i+1}, A), T_{cpyb}(P_{result,i}) \cdot FR(P_{result,i})) \\ & + \max(T_{comp}(P_n, A), T_{cpyb}(P_{n-1}) \cdot FR(P_{result,i})) + T_{cpyb}(P_n) \cdot FR(P_{result,i}). \end{aligned} \tag{6}$$

We now discuss the usage of the response time metric for the selection of the response time minimal sequential query plan in consideration of concurrent copying and processing.

6.4. Computing the response time of a hybrid query sequence

The estimated cost $T_{est}(QS_{hybrid})$ of a hybrid query QS_{hybrid} is the sum of all estimated execution times $T_{est}(A)$ for each algorithm A in QS_{hybrid} with respect to concurrent copying and processing if A is a GPU algorithm. The costs correspond to the response time of the operation sequence. Algorithm 2 outlines the computation of the response time. If a data transfer and a computation are concurrently processed, the

Algorithm 2 Computation of response time for hybrid query sequence

Input: QS_{hybrid}

Output: $T_{response}$ of QS_{hybrid}

```
1:  time=0
2:  for $A_i \in QS_{hybrid}$ do
3:      if $A_i == copyOperation$ then
4:          continue
5:      end if
6:      if $A_{i-1} == A_{cpy}$ then
7:          $A_{i-1}.D.NG = 1$
8:      else
9:          $A_{i-1}.D.NG = 0$
10:     end if
11:     if $A_{i+1} == A_{cpyb}$ then
12:         $A_{i+1}.D.FR = 1$
13:     else
14:         $A_{i+1}.D.FR = 0$
15:     end if
16:     time = time + $T_{response}(A_i)$
17: end for
```

flags FR (copy back to host in parallel) or NG (copy to device in parallel) are set to true or false. The flags are evaluated by the functions $FR(P)$ and $NG(P)$, where P is a part. Depending on the values, the data transfer time is part of the overall sum or not.

6.5. Data partitioning

We now address challenges for data partitioning, which have to be resolved. To be able to utilize metrics from this section, we have to support efficient partitioning of the data. We could use common partitioning schemes like range or hash partitioning. The problem is to choose the size of the parts. Larger parts mean less parts, which lead to better PCIe bus utilization but also to higher latency, before processing can start. Hence, it is not a trivial task to create a partition, which results in minimal processing time. Furthermore, data needs to be partitioned, if the data set is larger than the available GPU RAM. Note that some operations cannot be processed independently, e.g., sorting operations. A system can presort data parts, but the final sorting order must be determined by a global merge step on the whole data set. If multiple GPUs are available, it is beneficial to use them for query processing. If the data is already partitioned, the parts of the data set D can be distributed on a GPU and processed concurrently, which is likely to significantly decrease the query response time. We address this issue in future work.

7. The 2-copy-operation heuristic

Algorithm 3 Construction of QS_{hybrid} from QS_{log} using two copy heuristic

```
Input: $QS_{log} = (O_1, D^1); \cdots ; (O_n, D^n)$
Output: $QS_{hybrid} = A_1 \cdots A_m$
1: $T_{minimal\ resp} = \infty$
2: $QS_{hybrid} = \emptyset$
3: $QS_{hybrid\ min} = \emptyset$
4: for $i =; i < |QS_{log}|; i++$ do
5:    for $j =; j < |QS_{log}| - i; j++$ do
6:       $QS_{hybrid}$ =create_hybrid_query sequence _candidate($QS_{log}, i, j$)
7:       if $T_{resp}(QS_{hybrid}) < T_{minimal\ resp}$ then
8:          $T_{minimal\ resp} = T_{resp}(QS_{hybrid})$
9:          $QS_{hybrid\ min} = QS_{hybrid}$
10:      end if
11:   end for
12: end for
13: return $QS_{hybrid\ min}$
```

We already discussed the fact that the greedy hybrid query sequence construction algorithm is not optimal. Therefore, we present an optimization algorithm that uses the new cost metrics presented in Section 6 and that allows only two data transfers in a sequence. The refined approach is based on the observation of Gregg & Hazelwood (2011) that copy operations have significant overhead and GPU algorithms are often faster. Hence, it is very likely that an optimal hybrid query sequence contains a minimum of copy operations. Therefore, we allow at most two copy operations in one hybrid query sequence. That means, all hybrid query sequences of the form

$$A_{1,CPU}A_{2,CPU}\cdots A_{i,CPU}A_{cpy}A_{i+1,GPU}\cdots A_{j,GPU}A_{cpy}A_{j+1,CPU}\cdots A_{n,CPU}.$$

where $j > i, n \geq i \geq 1, n \geq j \geq 1$, are allowed. The allowed set of sequences also includes pure CPU plans as well as pure GPU plans. The 2-Copy-Operation heuristic reduces the optimization space from exponential in number of operations to quadratic in number of operations. Since the algorithms has to create a query plan for each point in the reduced optimization space, our optimization algorithm has cubic complexity in the number of operations, see Algorithm 3.

After initialization of local variables (lines 1–3), the algorithm traverses the optimization space using two nested loops. The algorithm tests all combinations of positions of data transfer algorithms (A_{cpy}, A_{cpyb}). That means, it changes the position and length of the GPU part

$$A_{cpy}A_{i+1,GPU}\cdots A_{j,GPU}A_{cpyb} = SubPlan_{GPU}(i, j)$$

Algorithm 4 Create hybrid query sequence candidate

Input: $QS_{log} = (O_1, D^1); \cdots ; (O_n, D^n), position, gpu\_sequence\_length$
Output: $QS_{hybrid} = A_1 \cdots A_m$

```
 1: $QS_{hybrid} = \emptyset$
 2: for $O_i$ in $QS_{log}$ do
 3:    if i<position or i>position+$gpu\_sequence\_length$ then
 4:       $A = CA_{CPU}(D^i, O)$
 5:    else
 6:       $A = CA_{GPU}(D^i, O)$
 7:    end if
 8:    $AS = CAS(A)$
 9:    append $AS$ to $QS_{hybrid}$
10: end for
11: //delete redundant copy operations
12: for $A_i$ in $QS_{hybrid}$ do
13:    if ($A_i = A_{cpyb}(D)$ and $A_{i+1} = A_{cpy}(D)$) then
14:       delete $A_i A_{i+1}$ from $QS_{hybrid}$
15:    end if
16: end for
17: return $QS_{hybrid}$
```

of the hybrid query sequence. The first loop changes the position of $SubPlan_{GPU}(i, j)$ in the query plan where as the second loop varies the length of $SubPlan_{GPU}(i, j)$ (lines 4–5). For every GPU sequence, the corresponding candidate plan is constructed by executing Algorithm 4 (line 6). The algorithm computes the response time of the candidate. The candidate is the current result if and only if the estimated response time of the query plan is lower than all previous observed candidate plans (lines 6–9). The response time is computed by Algorithm 2 that we introduced in Section 6. We consider possible concurrent data transfers and computation in this way. After completion of the loops, the minimal hybrid query sequence plan found is returned (line 13).

As already mentioned, Algorithm 4 creates a candidate plan for a logical query sequence and the position *position* and length *gpu_sequence_length* of the GPU part of the query. First, the algorithm initializes the candidate plan (line 1). Second, the algorithm traverses the logical query plan and chooses a GPU algorithm for operation O_i if i is greater than or equal the start position *position* of the GPU part and less than or equal the start position of the GPU part plus the length of the GPU part. Otherwise, a CPU algorithm is selected (line 3–7). Note that the functions $CA_{CPU}(D^i, O)$ and $CA_{GPU}(D^i, O)$ choose the best available CPU and GPU algorithm, respectively, using our decision model. In the next step, the function CAS is called and the returned algorithm sequence is added to the hybrid query sequence. As in Algorithm 1, the use of the function CAS may lead to redundant copy operations that have to be removed from the hybrid query plan (line 12–16). In the last step, the constructed candidate plan is returned (line 17).

Note that the 2-Copy-Operation heuristic is not guaranteed to find the response time minimal query plan. If the optimal plan uses more than two copy operations, the heuristic chooses a suboptimal plan. The 2-Copy-Operation heuristic considers the investigation of sequences of operations. In contrast, the greedy algorithm only uses local decisions. Therefore, it is more likely that the 2-Copy-Operation heuristic produces better hybrid query sequences than the greedy approach. However, the algorithm has a cubic time complexity compared to the linear time complexity of the greedy approach. Furthermore, the 2-Copy-Operation heuristic creates a quadratic number of candidate hybrid query sequences, while the greedy approach creates exactly one query sequence. We will investigate in future work, under what conditions, which algorithm is better.

8. Extension: query as tree of operations

We extend our discussed concepts and algorithms to support query trees using sequences as building blocks.

8.1. Optimization problem for query trees

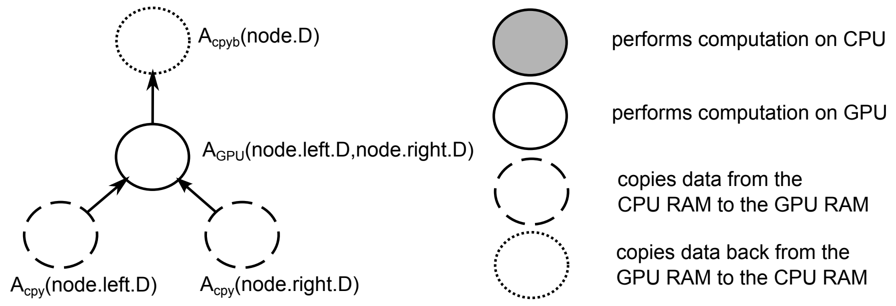
Similar as for query sequences, we have to remove redundant copy operations from a query tree. Therefore, we adapt our algorithms for sequences to trees. A tree node *node* is a 7-tuple (*id*, *name*, *parent*, *left*, *right*, *A*, *D*), where *id* is the unique identifier of the node, *name* is the name of the node, *parent*, *left*, *right* are the parent node, left and right child and *A* is the algorithm executed by the node (or Operation *O* for logical query tree). *D* is the result data set, after the algorithm of the node was executed.

For simplicity, we assume that neither the Critical Query Challenge, nor the Optimization Impact Challenge of the discussed challenges in Section 2.2 occur for a hybrid query tree. If the Copy Serialization Challenge or the Execution Time Prediction Challenge occur in a query tree, we can create the corresponding query sequence, because the operations in a query sequence are processed sequentially.

8.2. Constructing hybrid query trees

To optimize query trees, we redefine the functions $CA(D,O)$ and $T_{est}(A,D)$ and modify our algorithms.

Let $CA(D,O)$ be a function, which chooses the fastest algorithm A for a given data set D and an operation O. It uses the function T_{est} to compute estimated execution times for algorithms. T_{est} considers the time needed to transfer data to and from the GPU RAM in the case of a selected GPU algorithm. Hence, $CA(D,O)$ chooses a GPU algorithm only, if the execution time of a CPU algorithm is greater than the execution time of a GPU algorithm plus the time needed for the data transfers. Note that we can have two data transfers from the CPU to the GPU RAM, because we allow binary operations. Hence, they are considered in Equation (7). The *CAS* function is replaced by the function $CST(node)$ (create sub tree), which returns a sub tree needed



A_{cpyb}(node.D)
A_{GPU}(node.left.D,node.right.D)
A_{cpy}(node.left.D)
A_{cpy}(node.right.D)
performs computation on CPU
performs computation on GPU
copies data from the CPU RAM to the GPU RAM
copies data back from the GPU RAM to the CPU RAM

Figure 3. Example: subtree generated by algorithm 5

to execute algorithm A on the chosen processing device. In case of a CPU algorithm, $CST(node)$ returns a node where A is the selected algorithm. In the case of a GPU algorithm, $CST(node)$ returns a sub tree with tree levels. Depending on whether the Operation O is unary or binary, level 2 contains one node or two nodes, which execute copy operations from the CPU RAM to the GPU RAM using the A_{cpy} algorithm. The computation node is stored in level 1 and does the actual processing. It has the nodes in level 2 as its child nodes. If the computation node executes a unary operation, then the preceding copy node is the left child. The parent of the computation node is stored in level 0, which executes a copy operation from the GPU RAM to the CPU RAM using the A_{cpyb} algorithm. Fig. 3 displays an example subtree. Note that computation nodes are either white or gray, where white nodes denote a GPU algorithm and a gray node a CPU algorithm.

$$T_{est}(node,A) = \begin{cases} T_{est}(node.D,A) & \text{if } A = A_{CPU} \\ T_{est}(node.left.D,A_{cpy}) & \\ +T_{est}(node.right.D,A_{cpy}) & \\ +T_{est}(node.left.D, & \\ node.right.D,A_{GPU}) & \\ +T_{est}(node.D,A_{cpyb}) & \text{otherwise} \end{cases} \tag{7}$$

$$CA(node,O) = A \text{ with } T_{est}(node,A) = min\{T_{est}(node,A) | A \in AP_O\}. \tag{8}$$

We adapt Algorithm 1 for trees as follows. We stick to the principle to choose a GPU algorithm only if it is faster than a CPU algorithm including the copy overhead. However, we have to implement the function $CST(node)$, which replaces $CA(D,O)$, in algorithm 5. The algorithm returns the passed node (line 25), if it executes a CPU algorithm and constructs a subtree including copy operations for a node executing a GPU algorithm (lines 2–23). The algorithm takes care of creating the nodes and integrate them in the tree by updating the node pointers of *node*, the child nodes of *node* and the parent node of *node*.

Algorithm 5 ConstructSubtree(node)

```
Input: $Treenode : node$
Output: $Q_{phy}(Tree) forGPUalgorithminnode$
1: if node.A==A_GPU then
2:     leftchild = createNode($node.left.D$,$A_{cpy}$)
3:     leftchild.parent=node
4:     leftchild.left=node.left
5:     if node.left!=NULL then
6:         node.left.parent=leftchild
7:     end if
8:     node.left=leftchild
9:     if node.right!=NULL then
10:        rightchild = createNode($node.right.D$,$A_{cpy}$)
11:        rightchild.parent=node
12:        rightchild.right=node.right
13:        node.right.parent=rightchild
14:    end if
15:    node.right=rightchild
16:    newparent = createNode($node.D$,$A_{cpyb}$)
17:    if node.parent.left!=node then
18:        node.parent.left=newparent
19:    else
20:        node.parent.right=newparent
21:    end if
22:    newparent.parent=node.parent
23:    node.parent=newparent
24:    return newparent
25: else
26:    return node
27: end if
```

Algorithm 6 constructs a hybrid query tree plan from a logical query tree plan using the $CST(node)$ algorithm 5. First, the logical query tree is copied to a working copy, which will contain the final hybrid query tree (line 1). Second, the algorithm calls the *getLevelorder* function, which returns a queue that contains all nodes of the hybrid query tree. For each node in the queue, the algorithm calls decision model *CA* function, which returns the algorithm with lowest expected execution time and assigns the algorithm to the current node (lines 3–4). Afterwards, the algorithm uses the function $CST(node)$ to get an appropriate subplan. Since $CST(node)$ creates and integrates the subplan automatically into the hybrid query tree, the algorithm can ignore the return value.

After the algorithm created an initial hybrid query tree (lines 1–6), it has to remove redundant copy operations from the plan (lines 7–16). It, therefore, traverses the tree

Algorithm 6 Construct hybrid query tree for logical query tree

```
 1: $QT_{hybrid} = QT_{log}$
 2: queue = getLevelorder($QT_{hybrid}$)
 3: for all node in queue do
 4:     node.A= $CA$(node.left.D,node.right.D, $O$)
 5:     tmp = ConstructSubtree(node)
 6: end for
 7: for all node in $QT_{hybrid}$ do
 8:     if node.A==A$_{cpyb}$ and node.parent.A==A$_{cpy}$ then
 9:         //update pointer
10:         node.parent.parent.left=node.left
11:         node.left=node.parent.parent
12:         //delete unneccessary copy operations
13:         delete node.parent from $QT_{hybrid}$
14:         delete node from $QT_{hybrid}$
15:     end if
16: end for
17: return $QT_{hybrid}$
```

and deletes copy nodes if the current node uses an A_{cpyb} algorithm and the current nodes parent uses an A_{cpy} algorithm. We use the convention that if a node has a single child node, the child node is the left child of the node. Hence, the algorithm updates the left pointers of the parent and child nodes of the copy nodes (line 10–11). Afterwards, the copy nodes are deleted (line 13–14). As a last step, the algorithm returns the constructed hybrid query tree (line 17). Fig. 4 illustrates the algorithm for an example logical query tree.

For a hybrid query tree constructed by algorithm 6, the following three assertions have to be fulfilled. First, a white and a gray node must not be directly connected, there has to be at least one copy operation between them. Second, no redundant copy operations may occur in the plan. Third, at the end of the queries execution, the result data have to be in the CPU RAM. If assertion one or three are not fulfilled, the query plan is not executable. If only assertion two is not fulfilled, the plan is executable, but unlikely to be beneficial with respect to the response time optimization criterion.

8.3. Estimating the response time of query trees

We now modify our algorithms to be able to perform the cost computation for a hybrid query tree. The basic idea is to use the algorithm for the sequence queries to compute the response time of a hybrid query tree.

For simplicity, we disallow concurrent executions of operations on the GPU, because of the Execution Time Prediction Challenge. Additionally, we forbid concurrent copy operations in one direction, because of the Copy Serialization Challenge. Since the decision model already assigned estimated execution times for each algorithm in

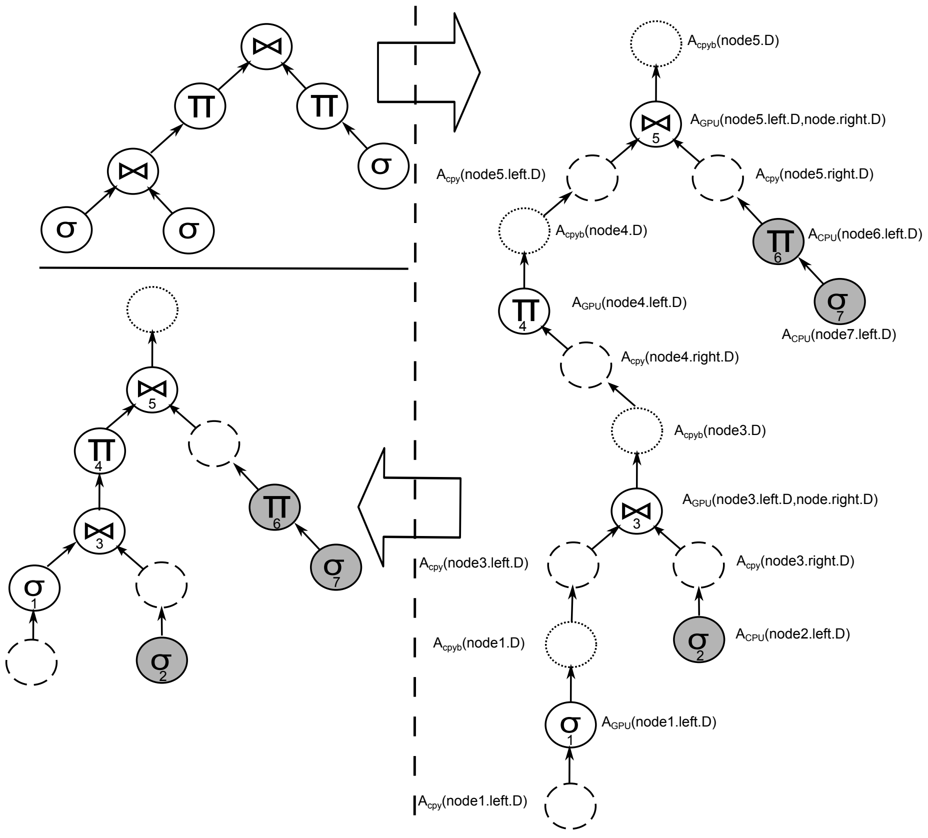


Figure 4. Example: constructing hybrid query tree

the hybrid query tree, we only need to find the critical path in the plan. Therefore, we have to create a sequence query for every possible path from the root node to one of the leave nodes of the hybrid query tree, which is done in algorithm 7. We apply our extended algorithm, which considers the overlapping of data transfer and computation, to each created path. The path with the highest response time dictates the lower bound of the response time of the hybrid query tree.

The upper bound is computed by turning the hybrid query tree into a hybrid query sequence and compute its response time. Then, the database optimizer can decide to use a hybrid query plan for execution or to use a different plan, e.g., a CPU only query tree. Note that our cost estimation algorithm can be used with other algorithms that construct hybrid query trees from logical query trees.

Algorithm 7 Computation of response time for hybrid query tree

Input: QT_{hybrid}
Output: $T_{response}$ of QT_{hybrid}

```
1: $T_{response}=-\infty$
2: for all node in $QT_{hybrid}$.getLeaves() do
3:     path = computePath(root,node)
4:     time = computeResponseTime(path) //considers concurrent data transfer and
       computation
5:     if time>$T_{response}$ then
6:         $T_{response}$=time
7:     end if
8: end for
9: return $T_{response}$
```

9. Related work

In this section, we will discuss related work. We discuss query optimization in a general context, other hybrid scheduling frameworks, learning based execution time estimation, and GPU co-processing.

9.1. Query optimization

Optimization in parallel database systems has similar tasks as optimization of GPU co-processing: optimizing the response time and scheduling operations to resources (Chaudhuri, 1998). Most approaches follow the two-phase optimization approach (Hong & Stonebraker, 1993). First, the database optimizer creates a best sequential query plan. Second, an additional optimizer allocates the operators to the parallel ressources to minimize the response time (Hasan, Florescu & Valduriez, 1996). Thereby, communication costs (Hasan, 1996) and different kinds of shared resources (Garofalakis & Ioannidis, 1997) have to be taken into account. Lanzelotte et al. (1994) noticed the enlarged search space and the problem of not optimal sub-plans during dynamic programming style enumeration. The authors showed that randomized search approaches during optimization have a good performance for parallel database systems. Our approach is also based on the two-phase model. We schedule a serial plan between GPU and CPU. Intra-operator parallelism is covered by the self-adaptive model (Breßet al., 2012; Breß, Mohammad & Schallehn, 2012). We focus on the communication costs between main memory and device memory in this work. We also have to consider the special situation that a GPU is a co-processor, and we do not have a symmetric system. For scheduling, adapted deterministic and randomized approaches are compared.

The parallelization of queries using threads of multi-core systems is also related. Krikellas, Cintra & Viglas (2010) used several greedy and dynamic programming approaches to schedule an operator tree on different threads to minimize the response time. Their approach is based on a symmetric environment and does not have to consider communication costs.

9.2. Hybrid scheduling frameworks

Ilić et al. (2011) showed that large benefits for database performance can be gained if the CPU and the GPU collaborate. They developed a generic scheduling framework (Ilić & Sousa, 2011), which is a similar approach to ours, but does not consider specifics of query processing. They applied their scheduling framework to databases and tested it with two queries of the TPC-H benchmark. However, they do not explicitly discuss hybrid query processing.

Augonnet et al. (2011) develop StarPU, which can distribute parallel tasks on heterogeneous processors. Both frameworks are extensible and have to be investigated to which degree they can be customized, so they can be used in a database optimizer. The biggest difference with our decision model is that it is tailor made for use in a database optimizer, so it provides, e.g., no task abstractions.

9.3. Learning based execution time estimation

Akdere & Cetintemel (2012) examined how analytical workloads can be modeled. Their approach can estimate execution times for single operations as well as queries and is based on feature extraction. Matsunaga & Fortes (2010) develop the PQR2 method, an approach to estimate the resource usage of applications. The approach can be used for execution time estimation, but needs several milliseconds to compute one estimation. This property makes it difficult to use the PQR2 method in a database optimizer. In contrast, we utilize the least squares method of the ALGLIB[1] for execution time estimation and observed execution times below 50 microseconds. Zhang et al. (2005) use the ”transform regression technique” to estimate the execution time of XML queries. Their approach a self-tuning optimizer similar to ours, but our goals and used statistical methods differ.

9.4. GPU co-processing

Current research investigates the use of GPUs for database operations (Bakkum & Skadron, 2010; He et al., 2009; Pirk, Manegold & Kersten, 2011; Walkowiak et al., 2010). Walkowiak et al. (2010) discuss the usability of GPUs for databases and show the applicability on the basis of an n-gram based text search engine. He et al. (2008, 2009) present the concept and implementation of relational joins on GPUs and of other relational operations.

Pirk, Manegold & Kersten (2011) develop an approach to accelerate indexed foreign key joins with GPUs. The foreign keys are streamed over the PCIe bus while random lookups are performed on the GPU. Furthermore, they introduce a new approach for GPU Coprocessing, which decomposes data bitwise. The approach uses

[1]`http://www.alglib.net/`

the GPU to process a low resolution version of the input data in a GPU preselection phase and then executes the CPU refinement phase, where the final results are computed by eliminating false positives from the result list (Pirk, 2012; Pirk et al., 2012). Hence, their approach tries to utilize CPU and GPU equally, similarly to our approach. However, our model balances the load on the operation level.

Kerr, Diamos & Yalamanchili (2010) present an approach that can select a CPU or a GPU implementation. In contrast to our decision model, their model decides for a CPU/GPU algorithm statically, whereas our decision model can do it dynamically. Their model does not introduce overhead at runtime.

Bakkum & Skadron (2010) develop a concept and implementation of the SQLite command processor on the GPU. The main target of their work is the acceleration of a subset of possible SQL queries. Govindaraju et al. (2004) present an approach to accelerate selections and aggregations with the help of GPUs.

He et al. (2009) developed a research prototype, which implements relational operations on CPU and GPU, respectively. They presented a co-processing scheme that assigns operations of a query plan to suitable processing devices (CPU/GPU). The developed cost model computes estimated execution times of single GPU algorithms in consideration of copy operations. They used a two-phase optimization model for queries. In the first phase, a relational optimizer creates an operator tree. In the second phase, the optimizer decides for every operator whether an operation is executed on GPU, CPU, or concurrently on both. He et al. (2009) proposed an exhaustive search strategy for small plans and a greedy strategy for large plans for the second phase. Since they used a calibration based method on top of an analytical cost model, their approach works currently for relational databases only, whereas our approach is more general and works with arbitrary algorithms, e.g., for XML databases. Our approach is also more general because the black-box self-adaptive mode allows the consideration of different load situations. From this research, we conclude that a GPU is an effective co-processor for database query processing.

Heimel (2011) created the prototype *Ocelot* by implementing GPU algorithms of common relational operations in MonetDB. He developed basic decision heuristics for choosing a processing unit for query execution. However, he did not consider hybrid query plans, where the CPU and the GPU are used to execute a query. Furthermore, Heimel identified two query optimizer problems. First, it is a necessity to have cost metrics, which enable the comparison of CPU and GPU algorithms. Second, the search space is bigger since placement of query plans (and hence operations) have many possibilities. Hence, he pointed out the need for a hybrid query processor and optimizer.

10. Future work

To address the problem of parallel processing of different queries, we will present a heuristic that will decide which database queries can benefit most from using the GPU, because not all queries can benefit from GPU co-processing.

An alternative approach to deal with parallelism within and between queries would be to allow both by default, and let the GPU schedule parallel requests on its own. As pointed out in Section 2.2, execution times will be harder to estimate and the benefit for single queries will decline. Nevertheless, our self-learning cost-estimation will adjust to this and can find a balance, because estimated execution times will increase due to concurrency situations. Furthermore, only queries benefiting most from a GPU-based execution will be executed as hybrid queries based on our described decision model. This approach has to be carefully evaluated.

Since our algorithm does not generate an optimal plan in all cases, other solutions have to be considered. Another approach to find the cheapest query plan would be to generate a candidate set of hybrid query plans, and apply our cost metrics to each of them and then choose the cheapest plan for execution. The possible benefit and overhead of this according approaches will be examined in future work. Furthermore, we will implement our framework in our prototype, which is a column oriented GPU accelerated DBMS.

11. Conclusion

In this paper, we pointed out common problems that occur during the optimization of hybrid query processing and need to be addressed to allow for an effective co-processing by the GPU during database query processing.

Furthermore, we provided a simple algorithm for constructing a good hybrid query sequence for a given logical query sequence using our scheduling framework and extended the algorithms and concepts for hybrid query trees. Additionally, we discussed cost metrics which consider concurrent processing and data transfer on GPU side to allow the optimizer to compute more realistic estimations for the response time of hybrid query sequences/trees.

12. Acknowledgement

The work in this paper has been partially funded by the German Federal Ministry of Education and Science (BMBF) through the Research Program under Contract No. FKZ: 13N10817. We thank Mario Pukall, Siba Mohammad as well as the reviewers of the Second ADBIS workshop on GPUs In Databases for helpful feedback and discussions.

13. References

Akdere, M. and Cetintemel, U. (2012) Learning-based Query Performance Modeling and Prediction. International Conference on Data Engineering (ICDE). IEEE, 390–401.

AMD Corporation (2011) *AMD Accelerated Parallel Processing OpenCL Programming Guide*, rev1.3f edition, Dec 2011.

Andrzejewski, W. and Wrembel, R. (2010) GPU-WAH: Applying GPUs to Compressing Bitmap Indexes with Word Aligned Hybrid. In *International Conferences on Database and Expert Systems Applications: Part II (DEXA (2))*. Springer, 315–329.

AUGONNET, C., THIBAULT, S., NAMYST, R. and WACRENIER, P.A. (2011) StarPU: a unified platform for task scheduling on heterogeneous multicore architectures. *Concurrency and Computation: Practice & Experience*, **23**(2), 187–198.

AUGUSTYN, D.R. and ZEDEROWSKI, S. (2012) Applying CUDA Technology in DCT-Based Method of Query Selectivity Estimation. In: *Second ADBIS workshop on GPUs In Databases (GID)*, Springer, 3–12.

BAGHSORKHI, S.S., DELAHAYE, M., PATEL, S.J., GROPP, W.D. and HWU, W.M.W. (2010) An Adaptive Performance Modeling Tool for GPU Architectures. *SIGPLAN Not.,*, **45**, 105–114.

BAKKUM, P. and SKADRON, K. (2010) Accelerating SQL database operations on a GPU with CUDA. In: *3rd Workshop on General-Purpose Computation on Graphics Processing Units*, GPGPU '10, ACM, 94–103.

BEIER, F., KILIAS, T. and SATTLER, K.U. (2012) GiST Scan Acceleration using Coprocessors. In: *Eighth Internationl Workshop on Data Management on New Hardware*, DaMoN'12, ACM, 63–69.

BREß, S., BEIER, F., RAUHE, H., SCHALLEHN, E., SATTLER, K.U. and SAAKE, G. (2012) Automatic Selection of Processing Units for Coprocessing in Databases. In: *16th East-European Conference on Advances in Databases and Information Systems (ADBIS)*, Springer, 57–70.

BREß, S., MOHAMMAD, S. and SCHALLEHN, E. (2012) Self-Tuning Distribution of DB-Operations on Hybrid CPU/GPU Platforms. In: *Grundlagen von Datenbanken (GvD)*, CEUR-WS, 89–94.

BREß, S., SCHALLEHN, E. and GEIST, I. (2012) Towards Optimization of Hybrid CPU/GPU query Plans in Database Systems. In: *Second ADBIS workshop on GPUs In Databases (GID)*, Springer, 27–35.

CHAUDHURI, S. (1998) An Overview of Query Optimization in Relational Systems. In: *Symposium on Principles of Database Systems (PODS)*, ACM, 34–43.

DIAMOS, G., WU, H., LELE, A., WANG, J. and YALAMANCHILI, S. (2012) Efficient Relational Algera Algorithms and Data Structures for GPU. Technical report, Center for Experimental Research in Computer Systems (CERS).

FANG, W., HE, B. and LUO., Q. (2010) Database Compression on Graphics Processors. *Proceedings of the VLDB Endowment (PVLDB)*, **3**, 670–680.

GAROFALAKIS, M.N. and IOANNIDIS, Y. (1997) Parallel Query Scheduling and Optimization with Time- and Space-Shared Resources. In: *3rd International Conference on Very Large Data Bases*, VLDB'97. Morgan Kaufmann Publishers Inc., 296–305.

GETOOR, L., TASKAR, B. and KOLLER, D. (2001) Selectivity estimation using probabilistic models. In: *International Conference on Management of Data*, SIGMOD'06, ACM, 325–336.

GOVINDARAJU, N., GRAY, J., KUMAR, R. and MANOCH, D. (2006) GPUTeraSort: High Performance Graphics Coprocessor Sorting for Large Database Management. In: *SIGMOD International Conference on Management of Data*, SIGMOD'06, ACM, 325–336.

GOVINDARAJU, N.K., LLOYD, B., WANG, W., LIN, M. and MANOCHA, D. (2004)

Fast Computation of Database Operations using Graphics processors. *SIGMOD International Conference on Management of Data*, SIGMOD '04, pages 215–226. ACM.

Gregg, C. and Hazelwood, K. (2010) Where is the data? Why You Cannot Debate CPU vs. GPU Performance without the Answer. In: *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software*, ISPASS'11, IEEEE, 134–144.

Hasan, W., Florescu, D. and Valduriez, P. (1996) Open Issues in Parallel Query Optimization. *SIGMOD Record*, **25**(3), 28–33.

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

He, B., Yang, K., Fang, R., Lu, M., Govindaraju, N., Luo, Q. and Sander, P. (2008) Relational Joins on Graphics Processors. In *SIGMOD International Conference on Management of Data*, SIGMOD '08, ACM, 511–524.

He, B., and Yu, J.X (2011) High-Throuhput Transaction Executions on Graphics Processors. *Proceedings of the VLDB Endowment (PVLDB)*, **4**(5), 314–325.

Heimel, M. and Markl, V. (2012) A First Step Towars GPU-assisted Query Optimization. In: *Third International Workshop on Accelerating Data Management Systems Using Modern Processor and Storage Architectures (ADMS'12).* www.adams-conf.org/heimel_adms12.pgf.

Heimel, M. (2011) Investigating Query Optimization for a GPU-accelerated Database. Master's thesis, Technische Universität Berlin, Electrical Engineering and Computer Science, Department of Software Engineering and Theoretical Computer Science.

Hong, S. and Kim, H. (2009) An Analytical Model for a GPU Architecutre with Memory-level and Thread-level Parallelism Awarness. *SIGARCH Comput. Archit. News*,, **37**152–163.

Hong, W. and Stonebraker, M. (1993) Optimization of Parallel Query Execution Plans in XPRS. *Distributed and Parallel Databases*, **1**(1), 9–32.

Ilić, A., Pratas, F., Trancoso, P. and Sousa, L. (2011) High Performance Scientific Computing with Special Emphasis on Current Capabilities and Future Perspectives. In: *High-Performance Computing on Heterogeneous Systems: Database Queries on CPU and GPU*, IOS Press, 202-222.

Ilić, A. and Sousa, L. (2011) CHPS: An Environment for Collaborative Execution on Heterogeneous Desktop Systems. *International Journal of Networking and Computing (IJNC)*, **1**(1).

Kaldewey, T., Lohman, G., Mueller, R. and Volk, P. (2012) GPU Join Processing Revisited. In: *Eighth International Workshop on data Management on New Hardware*, DaMoN'12, ACM, 55–62.

Kerr, A., Diamos, G. and Yalamanchili, S. (2010) Modeling GPU-CPU Workloads and Systems. In: *3rd Workshop on General-Purpose Computation on Graphics Processing Units*, GPGPU '10, ACM, 31–42.

Kothapalli, K., Mukherjee, R., Rehman, M.S., Patidar, S., Narayanan, P.J.

and SRINATHAN, K. (2009) A Perfromance Prediction Model for the CUDA GPGPU Platform. In: *International Conference on High Performance Computing (HiPC)*, IEEE, 463–472.

KIRKELLAS, M., CINTRA, M. and VIGLAS, S. (2010) Scheduling threads for ibntraquery parallelism on multicore processors. Technical Report EDI-INFR-RR-1345, University oof Edinburgh, School of informatics, http://www.inf.ed.ac.uk/publications/report/1345.html.

LANZELOTTE, R.S.G., VALDURIEZ, P., ZAÏT and ZIANE, M. (1994) Invited project review: Industrial-strength parallel query optimization: issues and lessons. *Inf. Syst.*, **19**(4), 311–330.

LAUER, T., DATTA, A., KHADIKOV, Z. and ANSELM, C. (2010) Exploring Graphics Processing Units as Parallel Coprocessors for Online Aggregation. In: *International Workshop on Data warehousing and OLAP, DOLAP'10*, ACM, 77–84.

MATSUNAGA, A. and FORTES, J.A.B. (2010) . On the Use of Machine Learning to Predict the Time and Resources Consumed by Applications. In: *International Conference on Cluster Cloud and Grid Computing*, 495–504. IEEE.

MOUSSALI, R., HALSTEAD, R., SAOOLUM, M., NAJJAR, W. and TSOTRAS, V.J. (2011) Efficient XML Path Filtering Using GPUs. In: *VLDB-Workshop on Accelerating Data Management Systems Using Modern Processor and Storage Architecutres (ADMS)*. www.adams-conf.org/p9-MOUSSALLI.pdf.

NIVIDIA (2012) NVIDIA CUDA C Programming Guide. http://developer.download.nvidia.com/compute/DevZone/docs/html/C/doc/CUDA_C_Programming_Guide.pdf, 30–34, Version 4.0, [Online; accessed 1-May-2012].

PIRK, H. (2012) Efficient Cross-Device Query Processing. *Proceedings of the VLDB Endowment*.

PIRK, H., MANEGOLD, S. and KERSTEN, M. (2011) Accelerating Foreign-Key Joins using Asymmetric Memory Channels. In: *VLDB - Workshop on Accelerating Data Management Systems Using Modern Processor and Storage Architectures (ADMS)*. VLDB Endowment, 585–597.

PIRK, H., SELLAM, T., MANEGOLD, S. and KERSTEN, M. (2012) X-Device Query Processing by Bitwise Distribution. In: *Proceedings of the Eighth International Workshop on Data Management on New Hardware*, DaMoN '12, 48–54. ACM.

SANDERS, J. and KANDROT, E. (2010) *CUDA by Example: An Introduction to General-Purpose GPU Programming*. Addison-Wesley Professional, 1st edition.

SCHAA, D. and KAELI, D. (2009) Exploring the Multiple-GPU Desing Space. In: *International Symposium on Parallel & Distributed Processing*, IPDPS'09. IEEE, 1–12.

WALKOWIAK, S., WAWRUCH, K., NOWOTKA, M., LIGOWSKI, L. and RUDNICKI, W. (2010) Exploring Utilisation of GPU for Database Applications. *Procedia Computer Science*, **1**(1), 505–513.

ZHANG, N., HAAS, P.J., JOSIFOVSKI, V., LOHMAN, G.M. and ZHANG, C. (2005) Statistical Learning Techniques for Costing XML Queries. In: *International Conference on Very Large Data Bases*, VLDB '05, VLDB Endowment, 289–300.

ZHANG, Y. and OWENS, J.D. (2011) A Quantitative Performance Analysis Model for GPU Architectures. *Computer Engineering*, 382–393.