Self-Tuning Distribution of DB-Operations on Hybrid CPU/GPU Platforms

Sebastian Breß
Otto-von-Guericke University Magdeburg
bress@iti.cs.uni-magdeburg.de

Siba Mohammad
Otto-von-Guericke University Magdeburg
siba.mohammad@st.ovgu.de

Eike Schallehn
Otto-von-Guericke University Magdeburg
eike@iti.cs.uni-magdeburg.de

ABSTRACT

A current research trend focuses on accelerating database operations with the help of GPUs (Graphics Processing Units). Since GPU algorithms are not necessarily faster than their CPU counterparts, it is important to use them only if they outperform their CPU counterparts. In this paper, we address this problem by constructing a decision model for a framework that is able to distribute database operations response time minimal on CPUs and GPUs. Furthermore, we discuss necessary quality measures for evaluating our model.

1. INTRODUCTION

In the context of database tuning, there are many different approaches for performance optimization. A new opportunity for optimization was introduced with General Purpose Computation on Graphics Processing Units (GPGPU) [12]. This approach allows to speed up applications that are suited for parallel processing with the help of GPUs. Parallel computing architectures like compute unified device architecture (CUDA) [12] make it possible to program a GPU almost as simple as a CPU. This technology opens a new branch in research that focuses on accelerating applications using GPUs.

CPUs are optimized for a low response time, meaning that they execute one task as fast as possible. The GPU is optimized for high throughput, meaning they execute as many tasks as possible in a fixed time. This is accomplished by massively parallel execution of programs by using multi threading. Furthermore, GPUs specialize on compute-intensive tasks, which is typical for graphics applications. Additionally, tasks with much control flow decrease performance on GPUs, but can be handled well by CPUs [12]. Consequently, database operations benefit differently by using the GPU. Aggregations are most suited for GPU usage, whereas selections should be outsourced with care. He et al. observed that selections are 2-3 times slower on the GPU compared with the CPU [6].

24^{th} GI-Workshop on Foundations of Databases (Grundlagen von Datenbanken), 29.05.2012 - 01.06.2012, Lübbenau, Germany.
Copyright is held by the author/owner(s).

1.1 New Research Trend

A new research trend focuses on speeding up database operations by performing them on GPUs [4, 6, 13, 14].

He et al. present the concept and implementation of relational joins on GPUs [6, 7]. Pirk et al. develop an approach to accelerate indexed foreign key joins with GPUs [13]. The foreign keys are streamed over the PCIe bus while random lookups are performed on the GPU. Walkowiak et al. discuss the usability of GPUs for databases [14]. They show the applicability on the basis of an n-gram based text search engine. Bakkum et al. develop a concept and implementation of the SQLite command processor on the GPU [4]. The main target of their work is the acceleration of a subset of possible SQL queries. Govindaraju et al. present an approach to accelerate selections and aggregations with the help of GPUs [5]. From the examples, we can conclude that a GPU can be an effective coprocessor for the execution of database operations.

1.2 Problem Statement

We assume that an operation is executed through an algorithm which uses either CPU or GPU. For which operations and data is it efficient to execute database operations on a GPU? A GPU algorithm can be faster or slower as its CPU counterpart. Therefore, the GPU should be used in a meaningful way to achieve maximal performance. That means, only if it is to be expected that the GPU algorithm is faster it should be executed.

We do not know a priori which processing device (CPU or GPU) is faster for which datasets in a given hardware configuration. We have to take different requirements into account to determine the fastest processing device:

- the operation to be executed,
- the size of the dataset to process,
- the processing power of CPU and GPU (number of processor cores, clock rate), and
- the current load condition on CPU and GPU.

The following contributions are discussed in the following:

1. A response time minimal distribution of database operations on CPUs and GPUs can achieve shorter execution times for operations and speedup query executions. Hence, a basic idea how such a model can be constructed is depicted in this paper.
2. For the evaluation of our model, we define appropriate quality measures.

To the best of our knowledge, there is no self-tuning decision model that can distribute database operations response time minimal on CPUs and GPUs.

The remainder of the paper is structured as follows. In Section 2, we present our decision model. Then, we discuss model quality metrics needed for evaluation in Section 3. Section 4 provides background about related work. Finally, we present our conclusion and future work.

2. DECISION MODEL

In this section, we present our decision model. At first, we discuss the basic model. Then, we explain details of the estimation and the decision components.

2.1 Basic Model

We construct a model that is able to choose the response time minimal algorithm from a set of algorithms for processing a dataset D. We store observations of past algorithm executions and use statistical methods to interpolate future execution times. We choose the algorithm with the smallest estimated execution time for processing a dataset.

Definitions: Let O be a database operation and let $AP_O = \{A_1, .., A_m\}$ be an algorithm pool for operation O, i.e., a set of algorithms that is available for the execution of O. We assume that every algorithm has different performance characteristics. Hence, the execution times of different algorithms needed to process a dataset D are likely to vary. A data set D provides characteristic features of the input data. In this paper, we consider only the data size. Other characteristics, e.g., data distribution or data types will be considered in future work.

Let $T_A(D)$ be the execution time of an algorithm to process the dataset D. We do not consider hardware specific parameters like clock rate and number of processor cores to estimate execution times. Instead, we learn the execution behavior of every algorithm. For this purpose, we assign to every algorithm A a learning method L_A and a corresponding approximation function $F_A(D)$. Let $T_{est}(A, D)$=$F_A(D)$ be an estimated execution time of algorithm A for a dataset D. A measured execution time is referred to as $T_{real}(A, D)$. Let a measurement pair (MP) be a tuple (D,$T_{real}(A, D)$). Let MPL_A be a measurement pair list, containing all current measurement pairs of algorithm A.

Statistical Methods: We consider the following statistical methods for the approximation of the execution behavior of all algorithms. The first one is the *least squares method* and the second is *spline interpolation* with cubic splines [3]. We use these approaches because we observe in our experiments minimal overhead and a good accuracy (relative error $< 10\%$) of the estimated execution times. While other methods can learn the execution behavior depending on more than one feature, they often need a large amount of time for updating approximation functions and computing estimations, see Section 4. In the case of the least square method $F_A(D)$ is a polynomial of degree n. In the case of cubic splines, $F_A(D)$ is a spline.

Abstraction: The execution time of algorithms is highly dependent on specific parameters of the given processing hardware. In practice, it is problematic to manage all parameters, so maintenance would become even more costly. Hence, we do not consider hardware specific parameters and let the model learn the execution behavior of algorithms using statistical method L_A and the corresponding approximation function $F_A(D)$.

Model Components: The model is composed of three components. The first is the algorithm pool AP_O which contains all available algorithm of an operation O. The second is the estimation component which computes execution time estimations for every algorithm of an operation. The third part is the decision component which chooses the algorithm that fits best for the specified optimization criteria. Depending on whether the chosen algorithm uses the CPU or the GPU, the corresponding operation is executed on the CPU or the GPU. In this paper, we only consider response time optimization. However, other optimization criteria like throughput can be added in future work. Figure 1 summarizes the model structure.

2.2 Estimation Component

This section describes the functionality of the estimation component. First, we discuss when the approximation functions of the algorithms should be updated. Second, we examine how the model can adapt to load changes.

2.2.1 Updating the Approximation Functions

For an operation O that will be applied on a dataset D the estimation component computes for each available algorithm of O an estimated execution time. For this, we need an approximation function that can be used to compute estimations. Since we learn such functions with statistical methods, we need a number of observations for each algorithm to compute the corresponding approximation functions. After we computed an initial function for each algorithm, our model is used to make decisions. Hence, the model operation can be divided into two phases. The first is the initial training phase. The second is the operational phase.

Since the load condition can change over time, execution times of algorithms are likely to change as well. Hence, the model should provide a mechanism for an adoption to load changes. Therefore, the model continuously collects measurement pairs of all executed algorithms of an operation.

There are two problems to solve:

1. **Re-computation Problem:** find a strategy when to re-compute the approximation function for each algorithm, so that a good trade-off between accuracy and overhead is achieved.

2. **Cleanup Problem:** delete outdated measurements pairs from a measurement pair list because the consideration of measurement pairs from past load conditions is likely to be less beneficial for estimation accuracy. Furthermore, every measurement pair needs storage space and results in higher processing time for the statistical methods.

There are different heuristics that deal with the re-computation of approximation functions. Zhang et al. present possible approaches in [15]. These are:

1. Re-compute the approximation function always after a new measurement pair was added to the measurement pair list.

2. Re-compute the approximation function periodically.

3. Re-compute the approximation function, if the estimation error exceeds a certain threshold.

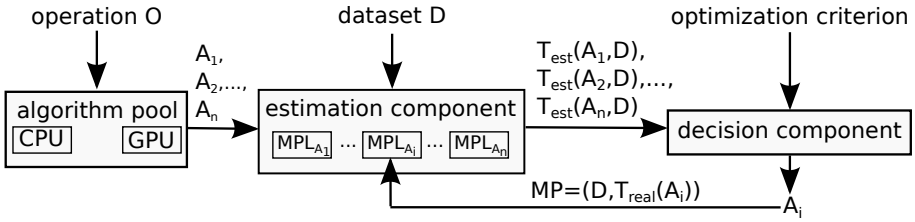

Figure 1: Information flow in the model

As Zhang et al. point out, the most aggressive method is unlikely to achieve good results because the expected overhead for re-computing the approximation function counteracts possible performance gains. Hence, we have to choose between approach 2 and 3. There is no guarantee that one approach causes less overhead than the other.

On one hand, periodic re-computation causes a predictable overhead, but it may re-compute approximation functions when it is not necessary, e.g., the relative estimation error is still beneath an error threshold. On the other hand, an event based approach re-computes the approximation functions only if it is necessary, but has the disadvantage, that the quality of estimations has to decrease until the approximation functions are re-computed. We choose the periodic approach, because of it's predictable overhead. We will consider the event based approach in future work.

Parameters: Now we discuss necessary parameters of the model. Let RCR be the re-computation rate of an algorithm. That means that after RCR measurement pairs were added to the measurement pair list of an algorithm A, the approximation function $F_A(D)$ of A is re-computed.[1] Hence, the used time measure is the number of added measurement pairs. Let ITL be the initial training length, which is the number of measurement pairs that have to be collected for each algorithm, before the model switches into its operational phase.

Now we address the *Cleanup Problem*. We have to limit the number of measurement pairs in the measurement pair list to keep space and computation requirements low. Furthermore, we want to delete old measurement pairs from the list that do not contribute to our current estimation problem sufficiently. These requirements are fulfilled by a ring buffer data structure. Let RBS be the ring buffer size in number of measurement pairs. If a new measurement pair is added to a full ring buffer, it overrides the oldest measurement pair. Hence, the usage of a ring buffer solves the *Cleanup Problem*. RCR and RBS are related because if RCR is greater than RBS there would be measurement pairs that are never considered for computing the approximation functions. Hence, RCR should be smaller than RBS .

Statistical Methods: The used statistical methods have to be computationally efficient when computing approximation functions and estimation values. Additionally, they should provide good estimations (relative estimation error <10%). Since the times needed to compute estimations sums up over time, we consider a method computationally efficient if it can compute one estimation value in less than 50μs. Our experiments with the ALGLIB [2] show that this is the case for the least squares and cubic splines.

[1]For each operation one measurement pair is added to the list of the selected algorithm.

Self Tuning Cycle: The model performs the following self tuning cycle during the operational phase:

1. Use approximation functions to compute execution time estimations for all algorithms in the algorithm pool of operation O for the dataset D.
2. Select the algorithm with the minimal estimated response time.
3. Execute the selected algorithm and measure its execution time. Add the new measurement pair to the measurement pair list MPL_A of the executed algorithm A.
4. If the new measurement pair is the RCR new pair in the list, then the approximation function of the corresponding algorithm will be re-computed using the assigned statistical method.

2.2.2 Adaption to Load Changes

This section focuses on the adaption to load changes of the model. We start with necessary assumptions, then proceed with the discussion how the model can adapt to new load conditions.

Let A_{CPU} be a CPU algorithm and A_{GPU} a GPU algorithm for the same operation O.

The basic assumptions are:

1. Every algorithm is executed on a regular basis, even in overload conditions. That ensures a continuing supply of new measurement pairs. If this assumption holds then a gradual adaption of the approximation functions to a changed load condition is possible.
2. A change in the load condition has to last a certain amount of time, otherwise the model cannot adapt and delivers more vague execution time estimations. If this assumption does not hold, it is not possible to continuously deliver good estimations.
3. The load condition of CPU and GPU are not related.

As long as the assumptions are fulfilled, the estimated execution time curves[2] approach the real execution time curves if the load changes. Increase and decrease of the load condition on the side of the CPU is symmetric compared to an equivalent change on the side of the GPU. For simplicity, but without the loss of generality, we discuss the load adaption mechanism for the CPU side.

[2]An execution time curve is the graphical representation of all execution times an algorithm needs to process all datasets in a workload.

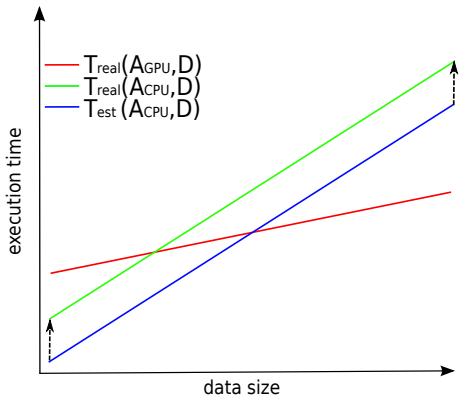

Figure 2: Example for increase load condition on CPU side

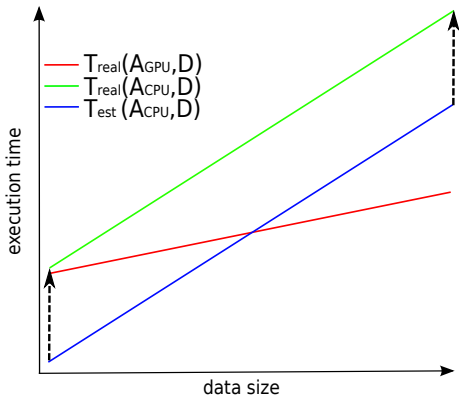

Figure 3: Example for strong increase of load on CPU side

Increasing Load Condition: If the load condition increases on the side of the CPU then the execution times of CPU algorithms increase and the real execution time curves are shifted upwards. In contrast, the estimated execution time curves stay as they are. We illustrate this situation in Figure 2. The model is adapted to the new load condition by collecting new measurement pairs and recomputing the approximation function. This will shift the estimated execution time curve in the direction of the measured execution time curve. Hence, the estimations become more precise. After at maximum RCR newly added measurement pairs for one algorithm the estimated execution time curve approaches the real execution time curve. This implies the assumption that a re-computation of approximation functions never has a negative impact on the estimation accuracy.

Decreasing Load Condition: The case of a decreasing load is mostly symmetric to the case of increased load. A decreased load on CPU side causes shorter execution time of CPU algorithms. The consequence is that real execution time curves are shifted downwards, whereas the estimated execution time curves stay as they are.

Limits of the Adaption: There is the possibility that the described load adaption scheme can break, if the load on CPU side increases or decreases too much. We consider the case, where the load increases too much, the other case is symmetric. In Figure 3, we illustrate the former case. Hence, the real execution time curve of A_{CPU} lies above the real execution time curve of A_{GPU}.

If this state continues to reside a certain amount of time, the estimated execution time curves will approach the real execution time curves. If the load condition normalizes again, then only algorithm A_{GPU} is executed, regardless of datasets that can be faster processed by algorithm A_{CPU}. The model is now stuck in an erroneous state since it cannot make response time minimal decisions for operation O anymore.

However, this cannot happen due to the assumption that every algorithm is executed on a regular basis, even in overload conditions. Hence, a continuing supply of new measurement pairs is ensured which allows the adaption of the model to the current load condition.

2.3 Decision Component

In this section, we describe the decision component of our model. Due to limited space, we introduce only one optimization criterion, namely response time, but other criteria like throughput are possible.

2.3.1 *Response Time Optimization*

If we optimize the operation execution for response time, we want to select the algorithm with the minimal execution time for the dataset D. In choosing the algorithm with the minimal execution time, we distribute the operation response time minimal to CPU and GPU. There is always one decision per operation that considers the size of the dataset that has to be processed.

Definition: Let a workload W be a tuple $W = (DS, O)$, where $DS = D_1, D_2, \cdots, D_n$ is a set of datasets D_i that are to be processed and O the operation to be executed.[3]

Goal: The goal is to choose the fastest algorithm $A_j \in AP_O$ for every dataset $D_i \in DS$ for the execution of operation O.

$$\text{min} = \sum_{D_i \in \text{DS}} T_{real}(A_k, D_i) \text{ with } A_k \in AP_O \tag{1}$$

Usage: The algorithm with the minimal estimated execution time for the dataset D is chosen for execution. The function choose_Algorithm (choose_Alg) chooses an algorithm according to the optimization criterion.

$$\text{choose\_Alg}(D, O) = A_j \text{ with} \tag{2}$$
$$T_{est}(A_j, D) = min(\{T_{est}(A_k, D) | \forall A_k \in AP_O\})$$

It is expected that the accuracy of the estimated execution times has a large impact on the decisions and the model quality.

Practical Use: The execution of the fastest algorithm reduces the response time of the system and results in better performance of a DBS. These response time optimization is necessary to accelerate time critical tasks[4]. Furthermore, it

[3]For simplicity, we only consider one operation per workload. However, a set of operations is more realistic and can be added in future work.

[4]A task is an operation in execution.

is possible to automatically fine-tune algorithm selection on a specific hardware configuration.

3. MODEL QUALITY CRITERIA

In this section, we present four model quality measures, namely average percentage estimation error, hit rate, model quality, and percentage speed increase.

3.1 Average Percentage Estimation Error

The idea of the average percentage estimation error is to compute the estimation error for each executed algorithm and the corresponding estimation value. Then, the absolute percentage estimation error is computed. Finally, the average of all computed percentage estimation errors is computed. This measure is also called relative error and is used, e.g., in [1].

3.2 Hit Rate

The idea of this measure is to compute the ratio of the number of correct decisions and the total number of decisions. A decision is correct, if and only if the model decided for the fastest algorithm. In the ideal case all decisions are correct, so the hit rate is 100%. In the worst case all decisions are wrong. Hence, the hit rate is 0%. The benefit of the hit rate is that it can provide statistical information about how many decisions are wrong. If the hit rate is X then every 1/(1-X) decision is incorrect. However, we cannot quantify the impact of an incorrect decision. This is addressed by the model quality. Note, that algorithms with very similar execution time curves can lead to a bad hit rate, although the overall performance is acceptable.

3.3 Model Quality

The idea of the model quality is to compute the ratio of the resulting execution times that a system using an ideal model and a real model needs to process a workload W. In the ideal case, the real model would be as good as the ideal model. Hence, the model quality is 100%. The worst case model would always select the slowest algorithm and provides a lower bound for the model quality.

Let $T_{DM}(W)$ be the time a system using a decision model (DM) needs to process a workload W. It is computed out of the sum of all algorithm execution times resulting from the algorithm selection of the decision model for the workload added with the overhead caused by DM. Let $T_{Oh}(DM, W)$ be the overhead the decision model DM causes. If a decision model introduces to much overhead, it can eliminate their gain. Hence the overhead has to be considered, which is the sum of the total time needed for computing estimation values and the total time needed to re-compute approximation functions. The time needed for all estimation value computation (EVC) for a workload W is $T_{\mathsf{EVC}}(W)$. The time needed to re-compute the approximation functions of all algorithms is $T_{RC}(W)$. Both measures are highly dependent on DM. Hence, we get the following formulas:

$$T_{Oh}(DM, W) = T_{\mathsf{EVC}}(W) + T_{RC}(W) \tag{3}$$

$$T_{DM}(W) = \sum_{D \in DS} T_{real}(\mathsf{choose\_Alg}_{DM}(D, O), D) + T_{Oh}(DM, W) \tag{4}$$

Let DM_{ideal} be the ideal decision model and let DM_{real} be the real decision model. Then, the model quality MQ is defined as:

$$MQ(W, DM_{real}) = \frac{T_{DM_{ideal}}(W)}{T_{DM_{real}}(W)} \tag{5}$$

This measure describes to which degree the optimization goal described in formula 1 is achieved. Note, that the ideal model is not introducing overhead ($T_{Oh}(DM_{ideal}, W) = 0$).

3.4 Percentage Speed Increase

The idea of the percentage speed increase (PSI) is to quantify the performance gain, if a decision model DM_i is replaced with a decision model DM_j.

$$PSI(DM_i \rightarrow DM_j, W) = \frac{T_{DM_i}(W) - T_{DM_j}(W)}{T_{DM_i}(W)} \tag{6}$$

If $PSI(DM_i \rightarrow DM_j, W)$ is greater than zero, DM_j had a better performance than DM_i and vice versa.

4. RELATED WORK

In this section, we present the related work. We consider analytical models for estimating execution times of GPU algorithms. Furthermore, we discuss learning based approaches to estimate execution times. Finally, we present existing decision models and provide a comparison to our approach.

Analytical Models Hong et al. present an analytical model which estimates execution times of massively parallel programs [8]. The basic idea is to estimate the number of parallel memory requests. The approach uses information like number of running threads and the memory bandwidth. Depending on their case study, relative estimation errors between 5,4% and 13,3% are observed.

Zhang et al. develop a model that should help optimizing GPU programs by arranging qualitative performance analyses [16]. They use a micro benchmark-based approach which needs hardware specific parameters, e.g., the number of processors or the clock rate. The model cannot adapt to load changes and therefore, it is not considered for use in our approach. The observed relative estimation error lies between 5% and 15%.

Learning based Execution Time Estimation Akdere et al. investigate modeling techniques for analytical workloads [1]. They estimate the execution behavior of queries on the basis of different granularities. They presented a modeling approach for the estimation on query level and operation level. The basic idea of their approach is to perform a feature extraction on queries and compute execution time estimations based on them.

Matsunaga et al. present a short overview over machine learning algorithms and their fields of application [11]. The goal is the estimation of the resource usage for an application. One considered resource is the execution time. The developed method PQR2 needs a few milliseconds for the computation of estimations. Since we need for n datasets and m algorithms n·m computations the time for a single computation should be less than 50μs to keep the overhead at an absolute minimum. Hence, the approach of Matsunaga et al. is to be investigated, whether it achieves good results

for a response time minimal operation distribution. This can be done in future work.

Zhang et al. present a model for predicting the costs of complex XML queries [15]. They use the statistical learning method called "transform regression technique". The approach allows a self-tuning query optimizer that can dynamically adapt to load condition changes. The approach of Zhang and our model are similar in basic functionalities. The difference to our approach is that Zhang et al. estimate the execution time of XML queries whereas our approach estimates execution times of single database operations to dynamically distribute them on CPU and GPU. Additionally, we use a different learning method that is optimized for computational efficiency.

Decision Models Kerr et al. present a model that predicts the performance of similar application classes for different processors [10]. The approach allows to choose between CPU and GPU implementation. This choice is made statically in contrast to our work, where an algorithm for an operation execution is chosen dynamically at runtime. Kerr et al. are using only parameters that are statically known before program execution. Hence, it allows no adaption to load changes in contrast to our model that allows load adaption.

Iverson et al. develop an approach that estimates execution times of tasks in the context of distributed systems [9]. The approach, similar to our model, does not require hardware specific information. They use the learning method k-nearest-neighbor, a non-parametric regression method. We use least squares and cubic splines that are parametric regression methods which need less time for computing estimations compared to non parametric regression methods. The goal of Iverson et al. is an optimal selection of nodes in a distributed system, where a task is executed. Unlike this approach, our work has the goal for optimal algorithm selection. It is possible to apply the approach of Iverson et al. on hybrid CPU/GPU platform. However, we consider, the GPU is a coprocessor of the CPU. Hence, a CPU/GPU platform is not a distributed system from our point of view. In a way, our model is less general than the model of Iverson et al.

5. CONCLUSION

In this paper, we addressed a current research problem, namely the optimal distribution of database operations on hybrid CPU/GPU platforms. Furthermore, we develop a self-tuning decision model that is able to distribute database operations on CPUs and GPUs in a response time minimal manner. We discuss the basic structure of the model and provide a qualitative argumentation of how the model works. Additionally, we present suitable model quality measures, which are required to evaluate our model. Our experiments show that our model is almost as good as the ideal model if the model parameters are set appropriately. We omit the evaluation due to limited space. We conclude that distributing database operations on CPUs and GPUs has a large optimization potential. We believe our model is a further step to address this issue.

In ongoing research, we use the defined quality measures to evaluate our model on suitable use cases. A further possible extension is using the model in a more general context, where response-time optimal decisions have to be made, e.g., an optimal index usage. Furthermore, an extension of the model from operations to queries is necessary for better applicability of our model. This leads to the problem of hybrid query plan optimization.

6. REFERENCES

[1] M. Akdere and U. Çetintemel. Learning-based query performance modeling and prediction. IEEE, 2012.

[2] ALGLIB Project. ALGLIB. `http://www.alglib.net/`, 2012. [Online; accessed 05-January-2012].

[3] P. R. Anthony Ralston. *A first course in numerical analysis*. dover publications, second edition, 2001.

[4] P. Bakkum and K. Skadron. Accelerating sql database operations on a gpu with cuda. GPGPU '10, pages 94–103, New York, NY, USA, 2010. ACM.

[5] N. K. Govindaraju, B. Lloyd, W. Wang, M. Lin, and D. Manocha. Fast computation of database operations using graphics processors. SIGMOD '04, pages 215–226, New York, NY, USA, 2004. ACM.

[6] 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.*, 34:21:1–21:39, December 2009.

[7] B. He, K. Yang, R. Fang, M. Lu, N. Govindaraju, Q. Luo, and P. Sander. Relational joins on graphics processors. SIGMOD '08, pages 511–524, New York, NY, USA, 2008. ACM.

[8] S. Hong and H. Kim. An analytical model for a gpu architecture with memory-level and thread-level parallelism awareness. *ACM SIGARCH Computer Architecture News*, 37(3):152, 2009.

[9] M. A. Iverson, F. Ozguner, and G. J. Follen. Run-time statistical estimation of task execution times for heterogeneous distributed computing. HPDC '96, pages 263–270, Washington, DC, USA, 1996. IEEE Computer Society.

[10] A. Kerr, G. Diamos, and S. Yalamanchili. Modeling gpu-cpu workloads and systems. GPGPU '10, pages 31–42, New York, NY, USA, 2010. ACM.

[11] A. Matsunaga and J. A. B. Fortes. On the use of machine learning to predict the time and resources consumed by applications. *CCGRID*, pages 495–504, 2010.

[12] NVIDIA. NVIDIA CUDA C Programming Guide. `http://developer.download.nvidia.com/compute/DevZone/docs/html/C/doc/CUDA_C_Programming_Guide.pdf`, 2012. pages 1–5, Version 4.0, [Online; accessed 1-February-2012].

[13] H. Pirk, S. Manegold, and M. Kersten. Accelerating foreign-key joins using asymmetric memory channels. ADMS '11, pages 585–597. VLDB Endowment, 2011.

[14] S. Walkowiak, K. Wawruch, M. Nowotka, L. Ligowski, and W. Rudnicki. Exploring utilisation of gpu for database applications. *Procedia Computer Science*, 1(1):505–513, 2010.

[15] N. Zhang, P. J. Haas, V. Josifovski, G. M. Lohman, and C. Zhang. Statistical learning techniques for costing xml queries. VLDB '05, pages 289–300. VLDB Endowment, 2005.

[16] Y. Zhang and J. D. Owens. A quantitative performance analysis model for gpu architectures. *Computer Engineering*, pages 382–393, 2011.