Information Systems 38 (2013) 1084–1096

Contents lists available at SciVerse ScienceDirect

Information Systems

journal homepage: www.elsevier.com/locate/infosys

Efficient co-processor utilization in database query processing[☆]

Sebastian Breß[a,*], Felix Beier[b], Hannes Rauhe[b,c], Kai-Uwe Sattler[b], Eike Schallehn[a], Gunter Saake[a]

[a] Otto von Guericke University Magdeburg, P.O. Box 4120, D-39016 Magdeburg, Germany
[b] Ilmenau University of Technology, P.O. Box 100 565, D-98684 Ilmenau, Germany
[c] SAP AG, Dietmar-Hopp-Allee 16, 69190 Walldorf, Germany

A R T I C L E I N F O

Article history:
Received 8 May 2013
Received in revised form
17 May 2013
Accepted 18 May 2013
Recommended by: D. Shasha
Available online 25 May 2013

Keywords:
Query optimization
Learning-based decision model
Database co-processing
Modern hardware architectures
In-memory databases

A B S T R A C T

Specialized processing units such as GPUs or FPGAs provide great opportunities to speed up database operations by exploiting parallelism and relieving the CPU. However, distributing a workload on suitable (co-)processors is a challenging task, because of the heterogeneous nature of a hybrid processor/co-processor system. In this paper, we present a framework that automatically learns and adapts execution models for arbitrary algorithms on any (co-)processor. Our physical optimizer uses the execution models to distribute a workload of database operators on available (co-)processing devices. We demonstrate its applicability for two common use cases in modern database systems. Additionally, we contribute an overview of GPU-co-processing approaches, an in-depth discussion of our framework's operator model, the required steps for deploying our framework in practice and the support of complex operators requiring multi-dimensional learning strategies.

1. Introduction

Recent trends in new hardware and architectures have gained considerable attention in the database community. Processing units such as *Graphics Processing Units* (GPU) or *Field Programmable Gate Arrays* (FPGA) provide advanced capabilities for massively parallel computation. Database processing can take advantage of such units not only by exploiting this parallelism, e.g., in query operators (either as task or data parallelism), but also by offloading computation from the *Central Processing Unit* (CPU) to these co-processors, saving CPU time for other tasks. In our work, we focus on *General Purpose Computing on GPUs* (GPGPU) and its applicability for database operations.

The adaption of algorithms for GPUs typically faces two challenges. First, the GPU architecture demands a fine-grained parallelization of the computation task. For example, Nvidia's Fermi GPUs consist of up to 512 thread processors, which are running in parallel lock step mode, i.e., threads execute the same instruction in an *Single Instruction Multiple Data* (SIMD) fashion on different input partitions, or idle at differing branches [2].

Second, processing data on a GPU requires data transfers between the host's main memory and the GPU's VRAM. Depending on each algorithm's ratio of computational complexity to I/O data volume this copy overhead may lead to severe performance impacts [3].

Thus, it is not always possible to benefit from massive parallel processing supported by GPUs or any other kind of co-processors. Assuming an efficient parallelization is implemented, break-even points have to be found where computational speedups outweigh possible overheads. To

[☆] This paper is an extended and revised version of an earlier work [1].
[*] Corresponding author. Tel.: +49 3916752845; fax: +49 3916712020.
E-mail addresses: sebastian.bress@ovgu.de, bress@iti.cs.uni-magdeburg.de, sebastianbress@gmx.de (S. Breß), felix.beier@tu-ilmenau.de (F. Beier), hannes.rauhe@tu-ilmenau.de (H. Rauhe), kus@tu-ilmenau.de (K.-U. Sattler), eike.schallehn@ovgu.de (E. Schallehn), gunter.saake@ovgu.de (G. Saake).

0306-4379/$ - see front matter © 2013 Elsevier Ltd. All rights reserved.
http://dx.doi.org/10.1016/j.is.2013.05.004

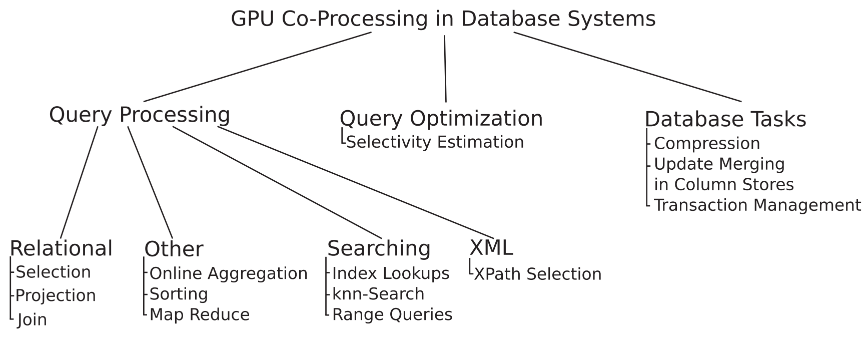

Fig. 1. Classification of co-processing approaches.

solve this scheduling decision, a system must be able to generate precise estimations of total processing costs, depending on available hardware, data volumes and distributions, and the system load when the system is actually deployed. This is further complicated by the rather complex algorithms which are required to exploit the processing capabilities of GPUs and for which precise cost estimations are difficult.

We address this challenge by presenting a self-tuning framework that abstracts from the underlying hardware platform as well as the actual task to be executed. It "learns" cost functions to support the scheduling decision and adapts them while running the tasks. We demonstrate the applicability of our approach on three problems typically faced in database systems which could benefit from co-processing with GPUs.

As extension to our prior work [1], we contribute a summary of co-processing approaches for databases and provide an in-depth discussion of the operator model that is used by the framework as well as its restrictions. Further, required steps for deploying it in practice are outlined and the model has been improved to support multi-dimensional parameters.

2. Co-processing approaches for databases systems

In the last decade GPUs became powerful and versatile enough to execute some general purpose calculations faster than CPUs. In this section, we contribute a short survey and classification of DBMS operations that can be offloaded to co-processors, especially the GPU. We introduce two important use cases in detail and show why we need automatic scheduling for these operations.

2.1. Co-processors in a DBMS

Fig. 1 puts the research in the field of database co-processing in three different classes, namely query processing, query optimization, and database tasks.

Query processing: We can find a plenitude of research that focuses on using GPUs and other co-processors to accelerate relational operators. Especially for joins there is a large variety of approaches for executing them on the GPU [4–6], on FPGAs [7], and even on Network Processing Units [8]. Other work also addresses the co-processing of all relational operators [9–11]. Index scan acceleration was investigated by Beier et al. [12] and Kim et al. [13]. Knn-search was studied by Wang et al. [14], Garcia et al. [15] and Barrientos et al. [16]. Spatial range queries were investigated by Pirk et al. [17]. There is also work addressing sorting [18], online aggregation [19] and XML path filtering [20]. There are several approaches addressing MapReduce, e.g., He et al. developed Mars [21].

Query optimization: Augustyn and Zederowski describe how to calculate the query selectivity estimation with a DCT algorithm on the GPU [22]. Heimel and Markl use kernel density estimation to estimate the query selectivity [23] and see this as a first step to optimize queries with the help of a co-processor.

Database tasks: There are more calculation intensive operations that are executed by the DBMS to maintain the stored data. Krüger et al. studied the process of merging the update buffer into the main storage of an In-Memory Column Store with the help of a GPU [24]. Data compression on GPUs was investigated in [25,26]. He et al. focused on transactional processing with graphic cards [27].

They all have in common that the operation or an essential part of it can be offloaded to the co-processor. Because of the offloading itself this involves some overhead, i.e., for small problem sizes the overhead often dominates the actually execution time. Also, the parameters of the operation and the data distribution change the calculation in a way that it does not fit to the co-processor's architecture anymore. Therefore, we cannot say that the operation is always faster on the co-processor than on the CPU counterpart and vice versa. Furthermore, without knowledge of the hardware, an a priori configuration is likely to be unfeasible and the user of the DBMS is not able to decide which version is best. Therefore, a hybrid scheduling framework is needed which chooses automatically the fastest algorithms depending on the operation to perform, its parameters, and the properties of data. Furthermore, it may be beneficial to utilize both, CPU and GPU, to increase throughput. Every use case can be seen as one operator, where we can decide to offload the calculation to a potential co-processor or execute it on the CPU. We choose two use cases and executed them as operator with the help of our framework.[1]

[1] In the non-extended version of this paper [1], another "Update Merge" use case was discussed. We left it out for the sake of space to discuss the model improvements compared to the framework of the original version.