Adaptive Reprogramming for Databases on Heterogeneous Processors

David Broneske
University of Magdeburg
Supervisor: Gunter Saake
david.broneske@ovgu.de

ABSTRACT

It is clear by now that modern processing hardware gets increasingly heterogeneous, which forces data processing algorithms to care about the underlying hardware. However, current approaches for implementing data intensive operators (e.g., in database systems) either cause enormous programming effort for tuning one algorithm to several processors (the hardware-sensitive way), or do not fully exploit possible performance possibilities because of an abstract operator description (the hardware-oblivious way). In this thesis, we propose an algorithm optimizer, which automatically tunes a hardware-oblivious operator description to the underlying hardware. This way, the DBMS can rewrite its operator code until it runs optimally on the given hardware.

Categories and Subject Descriptors

H.2 [**Information Systems**]: Database Management

General Terms

Design, Performance

Keywords

Heterogeneous Hardware; Adaptivity; Domain-specific Language; SIMD; Co-Processor Acceleration; Code Generation

1. INTRODUCTION

After decades of frequency scaling, single processors reach the end of performance improvements due to the power- and memory wall. A key to solve this problem is specialization and task distribution, which means that current systems get equipped with more and more specialized processors (e.g., FPGA, GPU, Intel Xeon Phi) which are optimized for a given task. While specialization yields performance opportunities for specific tasks, it also leads to an increased heterogeneity of processors in a system. This evolution in the hardware landscape forces database vendors and researchers to also specialize their algorithms for available processors [16].

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
SIGMOD'15 PhD Symposium, May 31, 2015, Melbourne, Victoria, Australia.
Copyright © 2015 ACM 978-1-4503-3529-4/15/05 ...$15.00.
http://dx.doi.org/10.1145/2744680.2744685.

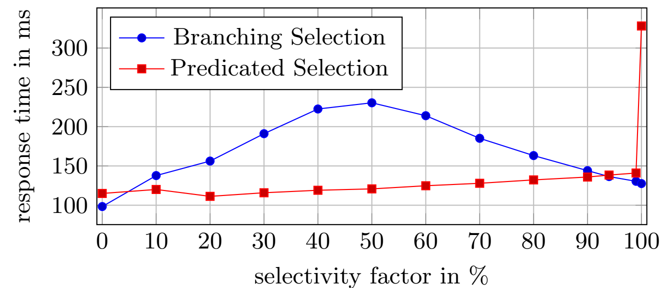

Figure 1: Scan variants on Intel Xeon E5-2690 – the best performing variant depends on the selectivity

For tuning a given algorithm to a processor, programmers mainly employ *code optimizations*. A code optimization transforms the code in order to improve its performance without changing its result. Prominent examples of code optimizations are loop fission, loop unrolling, predication (i.e., avoid branches in code), vectorization (i.e., using *Single Instruction Multiple Data* capabilities; SIMD), and parallelization [8, 25]. With the abstraction of code optimizations, we tune our algorithms to the used processor by choosing the right set of code optimizations. However, this is a non-trivial task, because the benefit of each code optimizations is not only depending on the processing capabilities of the processor, but also on the workload characteristics (e.g., selectivity or data size). To emphasize this statement, we show the response time of a normal database selection using an if-statement for evaluating the predicate and a predicated database selection without any branching behavior in Figure 1.

For very low and high selectivity factors, the branching version performs best while at medium selectivity factors, the predicated version performs best. The exact points where both lines cross depend on the used machine. Consequently, although code optimizations aim to improve the performance, they could also harm performance if either the hardware does not sufficiently support it, or the workload is unfavorable for this code optimization.

Investigating single code optimizations will make it easy to assess their benefit – especially for database management systems with their knowledge of stored data (e.g., selectivity estimation). However, it is difficult to assess the benefit of a combination of code optimizations, as we have shown in a former study [8]. As a consequence, we need a way

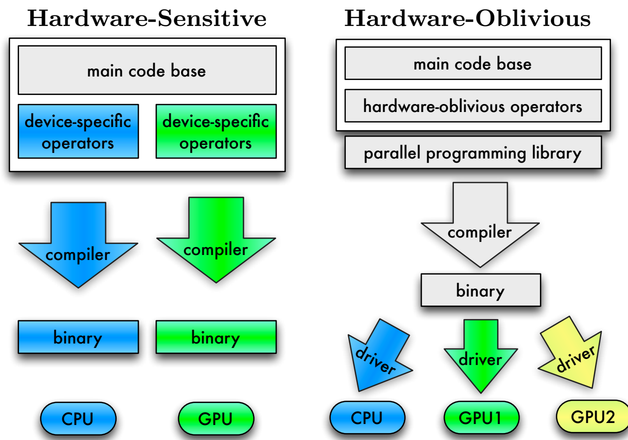

Figure 2: Hardware-sensitive vs. hardware-oblivious programming – adapted from [16]

to find the set of code optimizations that, applied to given code, produces the optimal code for the given machine and workload.

For solving the issue of finding the optimal operator for a given workload and machine, we propose a database management system that is exploring the code optimization space during query execution and automatically rewrites its own operator code. With this, we create a database management system that tunes its own database operators until the set of operators is optimal for the use case of the database system (i.e., workload and hardware).

In this paper, we first reflect current techniques to handle programming for heterogeneous hardware and present related work in Section 2. Then, we present the idea of an adaptive reprogramming approach in Section 3. In Section 4, we present the resulting research plan and present a small summary in Section 5.

2. HETEROGENOUS PROGRAMMING

To program for heterogeneous hardware, there are mainly two paradigms: the hardware-sensitive paradigm, in which the algorithms are tuned to one specific processor; or the hardware-oblivious paradigm, which means that the algorithms are abstractly defined and efficiently executed using a processor-dependent driver [16]. We depict a sketch of these two paradigms in Figure 2.

2.1 Hardware-Sensitive Programming

The main idea behind hardware-sensitive programming is that the programmer knows the system that the algorithm is written for in detail. So in a database, programmers would write a set of operators per processor and tailor the code to the underlying hardware by fully exploiting hardware's properties.

With this approach, programmers are able to reach the best performance, because they know what hardware to program for [16]. However, this approach does not scale to a high amount of different processors. The reason is that with each new processor, another set of operators has to be implemented, although they may only differ slightly. Thus, the development and maintenance effort is too high in this approach, especially if we have in mind the increasing heterogeneity of future processors.

Related Work

Despite the high development overhead, tuning operators to the underlying hardware has found much attraction in research. First, tuning focusses on optimizing main-memory database operators for different CPUs and the cache hierarchy.

CPU. Early work in this area includes to tune database selections using vectorization, e.g., the work of Zhou and Ross [30], or predication in the work of Ross [24]. Further optimizations for selections include vectorized scans on compressed data for single [27], and complex predicates [28], as well as using vectorized bloom filters for the scan [23].

Recent studies consider even more complex operators such as joins and aggregations. Here, Zukowski et al. optimize hash tables and functions to the underlying hardware [31]. Based on that, efficient vectorized aggregation functions are proposed by Polychroniou and Ross [22] to speed up aggregations in databases. Simultaneously, the debate about the best join algorithm has been revitalized, leading to even more specialized and tuned join algorithms. These include the *massively parallel sort-merge (MPSM) join* by Albutiu et al. [1], the sort-merge join using SIMD-accelerated sorting networks by Kim et al. [19], and the radix join which has been initially proposed by Boncz et al. [4] and further improved with vectorization and additional optimizations by Balkesen et al. [2, 3].

GPU. Early work considering database operations on GPUs has been published by He et al. [13], which uses highly optimized primitives on the GPU whose combination can compute any database operation. Furthermore, Sitaridi and Ross [26] present an efficient selection with GPU's missing branch-prediction capabilities. Also, several authors present how to adapt joins to GPUs [14, 18].

Other Processors. Furthermore, there is work presenting how to design database operations for more specialized co-processors. He et al. [15] show how to tailor the hash join to work efficiently on an APU (a CPU with an integrated GPU) and Jha et al. [17] investigate hash joins on an Intel Xeon Phi. Moreover, Mueller et al. [21] extensively discuss how to design sorting algorithms to implement them efficiently on FPGAs (field-programmable gate arrays).

All these publications show that tuning algorithms to the underlying hardware may improve performance by orders of magnitude. Nevertheless, all of them are only tailored to a single (co-)processor and do not provide a comprehensive solution for the increasing heterogeneity of the hardware landscape.

2.2 Hardware-Oblivious Programming

In contrast to hardware-sensitive programming, hardware-oblivious programming includes an additional abstraction layer: a parallel programming library. With this, database operations are implemented without explicit knowledge of the hardware based on the parallel programming library (e.g., OpenCL), which then compiles a binary for each processor [16]. This binary is executed using a specialized driver

for each processor which should exploit special hardware capabilities of the (co-)processor.

The advantage of hardware-oblivious programming is, that code for each operator is written only once and hardware-related properties are included by the driver. With this, development and maintenance overhead is reduced to a minimum. However, the compiler and driver optimize algorithms for the average use case and cannot take the workload into account. Furthermore, an efficient execution and exploitation of hardware capabilities always relies on a good implementation of the driver. Thus, it is not guaranteed that the hardware-oblivious approach always provides the best performance. Additionally, the driver is mainly designed to optimize for the general use case. With this, we are not able to fully exploit the domain knowledge that we have in database systems about the workload.

Related Work

Related work of hardware-oblivious programming mainly focusses on implementing a whole system with the new paradigm. For instance, *Ocelot*, the hardware-oblivious extension of MonetDB by Heimel et al. [16], maps an operator of MonetDB to a single implementation in OpenCL and reaches comparable performance on CPUs and even better performance on GPUs. Also, Zhang et al. [29] propose a DMBS for heterogeneous processors using OpenCL, called OmniDB. For this system, they propose a kernel-adapter based approach to be able to efficiently support different processors.

3. ADAPTIVE REPROGRAMMING

If we summarize the advantages of the above-mentioned paradigms, we reach the best performance using a hardware-sensitive approach, while having the best development and maintenance effort following a hardware-oblivious approach. Hence, the ideal solution would be a combination of both paradigms to maximize performance while minimizing programming effort. As a consequence, we propose adaptive reprogramming that is able to create several hardware-sensitive operators out of one abstract operator description. The overall structure of our approach is visualized in Figure 3.

3.1 Variant Generation

In our approach, we propose to use a domain-specific language (DSL) to define an operator [7]. With this, we add an abstraction level to the operator implementation, which helps us to flexibly adapt the resulting operator code. Given the generic operator description, we are able to apply different sets of code optimizations to produce different variants. For example, we could decide to vectorize the tight loop in a selection and then unroll the vectorized code.

The resulting variants are specific to one processor and have variations in code depending on different possible workloads. These variants are grouped per available processor type in a dedicated *variant pool*. On execution, the variant selector chooses the optimal variant for the selected processor and workload.

3.2 Variant Selector & Feedback Loop

Since it is not an easy task to estimate how a given algorithm performs on the given hardware – especially for parallel algorithms [2] – we argue to use a learning-based algorithm instead of a static cost model. The selector learns the

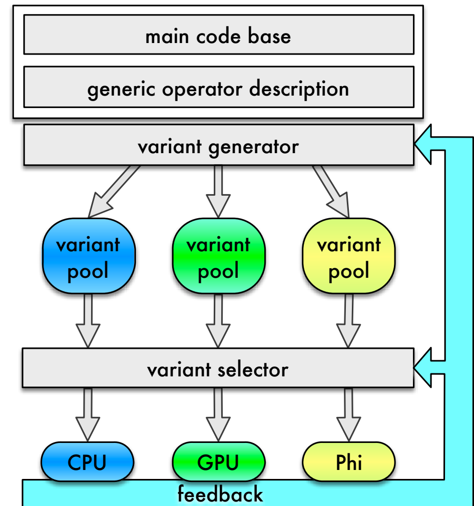

Figure 3: Sketch of our adaptive reprogramming approach for reaching hardware-sensitive database operations on heterogeneous hardware.

execution behavior of variants under given hardware properties and workload characteristics. In order to learn the execution behavior, we need a feedback loop, which informs the selector about the runtime of the chosen variant to refine the selector's learned cost model. Also, the feedback loop has to inform the variant generator about the performance impact of used code optimizations in order to generate more efficient code. This procedure resembles the idea of adaptive query processing [11], where the query plan is refined during run-time to achieve optimized performance.

3.3 Variant Management

In literature, there are numerous code optimizations proposed [8, 10, 25] that improve the code for different use cases. Hence, if we assume n independent code optimizations, we can create 2^n variants. Consequently, the number of possible variants increases exponentially with an increasing number of code optimizations. This causes the variant pool as well as the learned cost model of the selector to grow dramatically. As a consequence, we argue to limit the variant pool and to keep only promising variants in the pool. However, if the workload of the database system changes, better variants could be generated and included in the variant pool, while others are evicted.

4. RESEARCH PLAN

Arising from the proposed adaptive reprogramming approach for heterogeneous hardware, there are several areas that need to be investigated. Here, especially the influence of code optimizations, the design of the domain-specific language as well as the explorative approach for finding a promising set of new code optimizations play an important role. At the end, our approach helps also to further understand algorithm performance on modern processors.

4.1 Code Optimizations

At first, we investigate different code optimizations for different devices to show their benefits. In earlier work, we

have shown that for a simple scan on different CPUs, we can get large performance differences between the variants [8]. Furthermore, we expect much higher impacts for more complex operators such as joins and also for other processors such as GPUs or the Intel Xeon Phi.

For implementing our adaptive reprogramming approach, we choose CoGaDB, a column-oriented, GPU-accelerated DBMS [5], because it executes database operations on different (co-)processors. Furthermore, it already comes with a learning-based cost-estimator called HyPE [6], which allows us to learn the execution behavior of our variants.

4.2 DSL Design

We propose to use a domain-specific language for being able to automatically apply code optimizations to the resulting code. This DSL can vary in its abstraction level, i.e., from being very close to the C/C++ language to being rather functional. On the one hand, a low-level abstraction could make it possible to use specific compilers that may already have the ability to apply several code optimizations (e.g., the Polly LLVM compiler [12] allowing automatic loop parallelization and SIMD-ization).

On the other hand, an abstract definition allows for more coarse-granular refinement of the code. This approach has been shown in LegoBase, which uses a Scala DSL [20]. However, their approach so far shows how to optimize, e.g., the storage layout or the inter-operator execution, but leaves tuning to the processor open for future work. Furthermore, there is the Delite framework [9] which allows to build an own DSL and also provides several DSLs for programming for heterogeneous processors. Thus, these two approaches could be a good starting point to create several variants out of one abstract operator description by applying different code optimizations.

In order to argue the usefulness of the used DSL, we consider the flexibility of the DSL w.r.t. how many code optimizations are applicable. Thus, our goal is to design a DSL that is as flexible as possible, so that many code optimizations can be applied.

4.3 Exploration of Variant Space

An important aspect of the variant management is the creation of promising variants using a different set of code optimizations as before. A simple approach to create new variants is to generate one variant per code optimization and then combine the code optimizations of promising variants. However, with this approach, we resemble a greedy approach, which will probably find only a local optimum. A more mature approach is to use a heuristic approach such as simulated annealing or a genetic algorithm. For instance using a genetic algorithm, we represent the set of code optimizations as a feature vector. Those feature vectors are mutated and crossed to create new sets of code optimizations which generate new variants.

4.4 Understanding Variant Performance

At the end, our approach automatically generates several variants with different code optimizations enabled and keeps only promising variants active. Thus, the variants that are often used for a specific workload should be the optimal ones for the machine and workload. Hence, from the usage characteristic of the variants, we can conclude the suitability of specific code optimizations for the given machine and workload. Consequently, we expect that this thesis can further improve the understandability of algorithms on modern hardware.

5. SUMMARY

Driven by the increasing heterogeneity in the hardware landscape, in this thesis, we aim at an improved programming approach for implementing hardware-sensitive database operations for such heterogeneous hardware. For this, we analyzed current programming approaches for heterogeneous hardware and envision to combine the advantages of both approaches into a new one. In our adaptive reprogramming approach for heterogenous hardware, database operators are defined in a domain-specific language and, then, different variants for different (co-)processors are generated by applying a different set code optimizations. When a query is executed, a selector is choosing the right device and variant to be executed to suit the current workload and machine. With this approach, we reach peak performance for the given use case while minimizing development costs for the implemented operators. For implementing our proposal, we present research questions that have to be answered and present our research plan for this proposed thesis.

6. ACKNOWLEDGMENTS

I thank Gunter Saake for being my supervisor. Furthermore, I'd like to thank Sebastian Breß, Sebastian Dorok, Max Heimel, Andreas Meister, and Jens Teubner for their fruitful discussions and support.

7. REFERENCES

[1] M.-C. Albutiu, A. Kemper, and T. Neumann. Massively parallel sort-merge joins in main memory multi-core database systems. *VLDB*, 5(10):1064–1075, 2012.

[2] C. Balkesen, G. Alonso, J. Teubner, and M. T. Özsu. Multi-core, main-memory joins: Sort vs. hash revisited. *VLDB*, 7(1):85–96, 2013.

[3] C. Balkesen, J. Teubner, G. Alonso, and M. T. Özsu. Main-memory hash joins on multi-core CPUs: Tuning to the underlying hardware. *ICDE*, pages 362–373, 2013.

[4] P. Boncz, S. Manegold, and M. L. Kersten. Optimizing database architecture for the new bottleneck: Memory access. In *VLDB*, volume 9, pages 231–246. Springer, 1999.

[5] S. Breß. The design and implementation of CoGaDB: A column-oriented GPU-accelerated DBMS. *Datenbank-Spektrum*, 2014.

[6] S. Breß, F. Beier, H. Rauhe, K.-U. Sattler, E. Schallehn, and G. Saake. Efficient co-processor utilization in database query processing. *Inf. Sys.*, 38(8):1084–1096, 2013.

[7] D. Broneske, S. Breß, M. Heimel, and G. Saake. Toward hardware-sensitive database operations. In *EDBT*, pages 229–234, 2014.

[8] D. Broneske, S. Breß, and G. Saake. Database Scan Variants on Modern CPUs: A Performance Study. In *IMDM@VLDB*, pages 97–111. Springer, 2014.

[9] K. J. Brown, A. K. S. H. Lee, T. Rompf, H. Chafi, M. Odersky, and K. Olukotun. A heterogeneous

parallel framework for domain-specific languages. In *PACT*, pages 89–100, 2011.

[10] K. Datta, M. Murphy, V. Volkov, S. Williams, J. Carter, L. Oliker, D. Patterson, J. Shalf, and K. Yelick. Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures. In *SC*, pages 1–12, 2008.

[11] A. Deshpande, Z. Ives, and V. Raman. Adaptive Query Processing. *FTDB*, 1(1):1–140, 2007.

[12] T. Grosser, A. Groesslinger, and C. Lengauer. Polly - performing polyhedral optimizations on a low-level intermediate representation. *Parallel Processing Letters*, 22(04), 2012.

[13] B. He, M. Lu, K. Yang, R. Fang, N. K. Govindaraju, Q. Luo, and P. V. Sander. Relational query co-processing on graphics processors. In *TODS*, volume 34. pp. 21:1–21:39. ACM, 2009.

[14] B. He, K. Yang, R. Fang, M. Lu, N. K. Govindaraju, Q. Luo, and P. Sander. Relational joins on graphics processors. In *SIGMOD*, pages 511–524, 2008.

[15] J. He, M. Lu, and B. He. Revisiting co-processing for hash joins on the coupled CPU-GPU architecture. *VLDB*, 6(10):1–14, 2013.

[16] M. Heimel, M. Saecker, H. Pirk, S. Manegold, and V. Markl. Hardware-oblivious parallelism for in-memory column-stores. *VLDB*, 6(9):709–720, 2013.

[17] S. Jha, M. Lu, X. Cheng, B. He, and H. P. Huynh. Improving main memory hash joins on Intel Xeon Phi processors: An experimental approach. *VLDB*, 8(6), 2015.

[18] T. Kaldewey, G. Lohman, R. Mueller, and P. Volk. GPU join processing revisited. In *DaMoN*, pages 55–62. ACM, 2012.

[19] C. Kim, E. Sedlar, J. Chhugani, T. Kaldewey, A. D. Nguyen, A. D. Blas, V. W. Lee, N. Satish, and P. Dubey. Sort vs. hash revisited: Fast join implementation on modern multi-core CPUs. *VLDB*, 2(2):1378–1389, 2009.

[20] Y. Klonatos, C. Koch, T. Rompf, and H. Chafi. Building efficient query engines in a high-level language. *VLDB*, 7(10):853–864, 2014.

[21] R. Mueller, J. Teubner, and G. Alonso. Sorting networks on FPGAs. *VLDB Journal*, 21(1):1–23, June 2011.

[22] O. Polychroniou and K. A. Ross. High throughput heavy hitter aggregation for modern SIMD processors. In *DaMoN*, pages 37–42, 2013.

[23] O. Polychroniou and K. A. Ross. Vectorized bloom filters for advanced SIMD processors . In *DaMoN*, pages 49–54, 2014.

[24] K. A. Ross. Selection conditions in main-memory. *TODS*, 29(1):132–161, 2004.

[25] B. Răducanu, P. Boncz, and M. Zukowski. Micro adaptivity in vectorwise. In *SIGMOD*, pages 1231–1242, 2013.

[26] E. A. Sitaridi and K. A. Ross. Optimizing select conditions on GPUs. In *DaMoN*, pages 22–30, 2013.

[27] T. Willhalm, Y. Boshmaf, H. Plattner, N. Popovici, A. Zeier, and J. Schaffner. SIMD-Scan: Ultra fast in-memory table scan using on-chip vector processing units. *PVLDB*, 2(1):385–394, 2009.

[28] T. Willhalm, I. Oukid, I. Müller, and F. Faerber. Vectorizing database column scans with complex predicates. In *ADMS*, pages 1–12, 2013.

[29] S. Zhang, J. He, B. He, and M. Lu. OmniDB: Towards portable and efficient query processing on parallel CPU/GPU architectures. *VLDB*, 2(1):1374–1377, 2013.

[30] J. Zhou and K. A. Ross. Implementing database operations using SIMD instructions. In *SIGMOD*, pages 145–156, 2002.

[31] M. Zukowski, S. Héman, and P. Boncz. Architecture-conscious hashing. In *DaMoN*, pages 41–49, 2006.