Toward Hardware-Sensitive Database Operations

David Broneske
University of Magdeburg
Germany
dbronesk@ovgu.de

Sebastian Breß
University of Magdeburg
Germany
sebastian.bress@ovgu.de

Max Heimel
Technische Universität Berlin
Germany
max.heimel@tu-berlin.de

Gunter Saake
University of Magdeburg
Germany
saake@ovgu.de

ABSTRACT

Satisfying the performance needs of tomorrow typically implies using modern processor capabilities (such as *single instruction, multiple data*) and co-processors (such as *graphics processing units*) to accelerate database operations. Algorithms are typically hand-tuned to the underlying (co-)processors. This solution is error-prone, introduces high implementation and maintenance cost and one implementations is not portable to other (co-)processors. To this end, we argue for a combination of database research with modern software-engineering approaches. We emphasize our vision of generating optimized database algorithms tailored to used (co-)processors from a common code base. With this, we maximize performance while minimizing implementation and maintenance effort of hardware-tailored database operations.

1. INTRODUCTION

The growing amount of data is forcing modern database systems to cope with an increasing demand to deliver improved query performance. Historically, Moore's Law allowed us to improve the performance of a database server by simply upgrading its hardware from time to time – a process also called *vertical scaling* [20]. However, the last few years have shown that frequency scaling is significantly slowing down due to limitations arising from, e.g., the power wall[1] and the memory wall[2]. Instead, hardware vendors increasingly rely on specialization and diversification to improve performance; a process that will likely lead to a significantly more heterogeneous hardware landscape in the future [20]. Already today, highly specialized co-processors (e.g., *graphics cards* or *field-programmable gate arrays*) are frequently found in database servers and have been successfully used to improve query performance [11, 14, 21].

Unfortunately, in order to provide optimal performance, these specialized architectures usually require algorithms that are specifically tailored to the processing device. Since the behavior and performance of algorithms – especially when parallelized – is non-trivial to predict, devising such a system is a very challenging task. Thus, more and more researchers demand a proper understanding of algorithm's performance on modern hardware [1, 4], leading us to the following problem statement.

1.1 Problem Statement

The use of heterogeneous processing devices requires hand-tuned algorithms to exploit hardware capabilities and develop database management systems with the best performance. To achieve maximized performance, hand-tuned algorithms have to be developed without implying high development and maintenance cost.

1.2 Our Vision

Our vision is to combine the advantages of hand-tuned database algorithms with the expressiveness of high-level programming languages to maximize the performance of database operations on arbitrary hardware from a single code base. Our goal is to transfer recent research results from the software-engineering domain to the database domain. To this end, we propose an interdisciplinary solution for the problem of mapping impact factors of new hardware to database operations [9].

The remainder of this paper is structured as follows. In Section 2, we introduce prominent processing devices and review important processor capabilities of each device in Section 3. Arising from the heterogeneity of processing devices, we contribute our vision of hardware-tailored database operations in Section 4 and conclude in Section 5.

2. CURRENT PROCESSING DEVICES

In the literature, there is a steady improvement of algorithms for different database operations. Every time a new processing device is introduced, algorithms are tuned to new hardware features such as *AVX* [22] or *SSE extensions* [24]. In this section, we review common processing devices used for database operations and outline important capabilities of the devices. Current processing devices include *multi-core CPUs*, *GPUs*, *MICs*, *APUs*, and *FPGAs*.

[1]To achieve better clock rates, more power would be needed resulting in higher heat dissipation that cannot be handled.
[2]Access times of main memory have not evolved as good as processor capabilities of CPUs, which leads to an underutilization of CPUs.

10.5441/002/edbt.2014.22

Multi-Core Central Processing Unit.

The *central processing unit (CPU)* is the main processor of the computer. Since a multi-core CPU consists of several cores per chip supporting several threads per core, a high parallelism is achievable to execute computation-intensive tasks [17]. However, parallelism is only helpful if the code is parallelizable. Otherwise, the CPU pipelines sequential code to decrease idle time [6]. For this, the CPU supports branch prediction to evaluate `if`-statements and insert the probably right instructions in the pipeline. Another important capability is to execute SIMD instructions (*single instruction, multiple data*), which enables data parallelism. Here, one instruction is executed on multiple data items in one cycle, which decreases cache misses and improves instruction throughput.

Graphics Processing Unit.

With the introduction of general-purpose computing on *graphics processing units* (GPUs), a growing attention focuses on GPUs as processing devices for database operations [10, 12]. GPUs typically have dedicated high-bandwidth memory. Nevertheless, input data have to be transferred to the GPU RAM and result data have to be copied back to CPU RAM. Once data is accessible, GPUs offer highly-efficient SIMD capabilities executed on a large number of cores. However, since several GPU cores share the same instruction decoder, threads executing different branches have to be serialized.

Many Integrated Core.

Arising from the benefits of GPUs, the idea to pack a number of cores on one separate processing device became attractive. Hence, Intel proposed their *many integrated core (MIC)* architecture [20]. Their MIC, the Xeon Phi, contains around 60 Xeon processors packed in one dedicated device [20]. Each core supports up to four hardware threads and a similar memory hierarchy as traditional CPUs. However, L1- and L2-caches are shared between cores allowing fast access to data of neighboring cores. Since a MIC consists of several autonomous CPUs, it offers massive thread parallelism. Furthermore, each processor contains 512-bit vector processing units to provide high data parallelism in the execution. Similar to the GPU, a MIC can process only data dormant in its device memory.

Accelerated Processing Unit.

The so-called *accelerated processing unit (APU;* also *coupled CPU-GPU architecture)* consists of a CPU with an integrated GPU. Consequently, an APU unites properties of both devices. Furthermore, the CPU and GPU in an APU use the same main memory as well as consolidated caches, which is a high benefit in contrast to autonomous CPU and GPU, because of reduced data transfer cost [13]. However, since chip space is limited, the GPU part in the APU has less cores and runs on a lower frequency than their autonomous counterparts.

Field-Programmable Gate Array.

The *field-programmable gate array (FPGA)* represents a novel processing device for database operations. An FPGA consists of a number of *look-up tables* and *interconnects* between them. A look-up table can take an arbitrary boolean function and can be re-programmed at any time (with some re-programming delay). With this, logic units such as those used in CPUs can be emulated and tailored to the application. FPGAs are well suited for pipeline parallelization, because logic units can be pipelined to execute several functions in one clock cycle. Furthermore, FPGAs can be programmed to offer data parallelism, because the logic units of one function can be replicated on the chip. However, the chip space is limited and, hence, data parallelism is limited as well [21].

3. CAPABILITIES OF MODERN PROCESSING DEVICES

In this section, we emphasize important properties of processing devices that influence the performance of database operations. Here, we consider properties concerning *parallelization* and *memory scaling*.

3.1 Parallelization Aspects

Considering the parallelization in modern processing devices, we distinguish between *pipeline parallelism* and *data parallelism*. In the following, we characterize each property briefly and show which processing device offers the best support for each property.

Pipeline Parallelism.

Pipeline parallelism from the software point of view means that if for a number of data items, several instructions have to be executed one after another, the instructions can be pipelined. With this, instructions are prefetched to reduce instruction cache miss latency. Essentially, when an `if`-clause occurs, the device has to be able to guess the correct code branch. To this end, especially the CPU features branch prediction to estimate the correct branch based on previous executions of this particular branch. However, if a branch is executed as often as it is not executed (e.g., in selections with selectivity factor 0.5), branch mispredictions are bound to occur, which causes delays for processing [24].

Depending on the given code and problem, the CPU offers good to excellent pipelining capabilities, when branch prediction offers reliable results. Since the APU is a CPU with integrated GPU and since the MIC consists of several CPUs, they also provide good to excellent pipeline parallelism. In contrast, the GPU supports pipeline parallelism poorly, because little chip space is spent on control logic. In case of diverging branches, thread execution on GPU cores are serialized. Excellent pipelining capabilities are offered by an FPGA, because when including pipeline registers between each logic unit, data items can be shifted through pipelined instructions at each clock cycle [21]. Furthermore, pipelines are programmed into the FPGA, which eliminates pipeline flushes due to branch misprediction.

Data Parallelism.

With data parallelism, one instruction can be applied to multiple data items simultaneously. Although this reduces data cache misses, an important requirement for data parallelism is that data items can be processed independently from each other, because data has to be partitioned for parallelization.

Considering CPUs, the introduction of AVX- or SSE-registers enables them to offer a good capability to process several data items in parallel. However, their ability is still less powerful compared to GPUs and MICs. Furthermore, APUs support data parallelism well, although they cannot compete with dedicated GPUs, because of the limited chip

Processing Device	Parallelization Properties		Memory Scaling	
	Pipeline Parallelism	Data Parallelism	Memory Capacity	Memory Bandwidth
Central Processing Unit	+ / + +	+	+ +	+
Accelerated Processing Unit	+ / + +	+	+ +	+
Many Integrated Core	+ / + +	+ / + +	+	+ +
Graphics Processing Unit	–	+ / + +	+	+ +
Field-Programmable Gate Array	+ +	– / +	– / +	+ +

Legend: + + = excellent, + = good, – = poor

Table 1: Processing devices and their processor capabilities.

space on an APU limits the number of cores. MICs and GPUs have good to excellent data-parallelism capabilities, because they offer several cores for several partitions of data. They are only limited to the independence of operations on processed data items. In FPGAs, there is a trade-off between data and pipeline parallelism and may be configured by the user as needed. In fact, pipeline parallelism is more beneficial on an FPGA than data parallelism [21]. Thus, an FPGA offers poor to good data-parallelism capabilities depending on the configuration.

3.2 Memory Scaling

We identify two important properties influencing memory scaling capabilities which are *memory capacity* and *memory bandwidth*.

Memory Capacity.

The memory capacity is important for data intensive applications. Dedicated devices tend to have less memory than those that are able to use the main memory for processing. Furthermore, data has to be present at the accessible memory of the device to allow processing. Hence, if data does not fit in local memory, data processing is slowed down.

Considering our processing devices, CPU and APU are able to use the main memory. Furthermore, when using *non-uniform memory access (NUMA)* capabilities of these processing devices, the installable RAM can be extended further allowing them an excellent memory capacity of several hundreds to thousands of gigabytes of available memory. In contrast, since the GPU and MIC are dedicated devices, they use their own memory which is only several tens of gigabytes. Thus, we still account them a good memory capacity. The FPGA represents a special case, because it is usually processing streams of data. Thus, depending on the installed FPGA, it offers poor to good memory capacity.

Memory Bandwidth.

With memory bandwidth, we describe the speed of access on data which is already located on the working memory of the device. This property has gained importance, because memory bandwidth is the new bottleneck for query processing [5]. Here, especially NUMA capabilities have to be considered, because access to the memory of neighboring CPU-sockets is slower and should be avoided [16].

The bandwidth inside an FPGA, MIC and GPU is excellent, because the memory is hard-wired to its processing units. In contrast, CPUs and APUs still offer a good memory bandwidth although extending memory capacity using NUMA poses new challenges for database systems [16].

Conclusion.

Parallelization and memory capabilities differ strongly between processing devices as we summarize in Table 1. Since these properties influence the performance of database operations, their impact has to be considered in the implementation process of database operations.

4. TOWARDS TAILOR-MADE DATABASE OPERATIONS

Database algorithms have to be tailored to the capabilities of modern processing devices to exploit their full potential. To this end, first, we identify implementation stages in which programmers have to take care of the processor capabilities. Second, we motivate our vision how to produce hardware-sensitive database operations in a maintainable way. Third, we reinforce our vision by a possible solution.

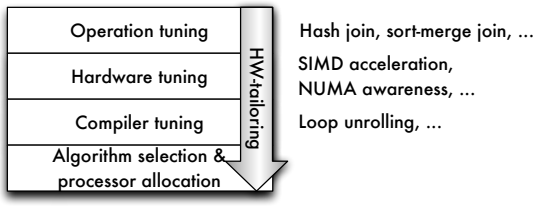

Figure 1: Tailoring database operations.

4.1 Implementation Phases

The implementation of hardware-sensitive database operations has to be done in different phases. We identify four phases of tailoring, which we visualize in Figure 1.

As an initial step, different algorithms for a database operation are implemented. Here, several authors have already proposed different algorithms for database operations [2, 4, 12]. After that, each algorithm has to be tailored to available processing devices. Thus, for each combination of algorithm and processing device, one *flavor* (a slightly modified algorithm [19]) is created. The third step includes compiler tuning, because Răducanu et al. identify that the choice of the compiler also influences algorithm performance [19]. As the last step, the database system has to be aware of available flavors of algorithms for different processing devices. Thus, rules for algorithm selection and processor allocation have to be adapted to conform to available algorithms and hardware [8, 7].

Each step has already been covered more or less by other researchers. However, a comprehensive solution considering each phase is missing. More severely, there is a huge number

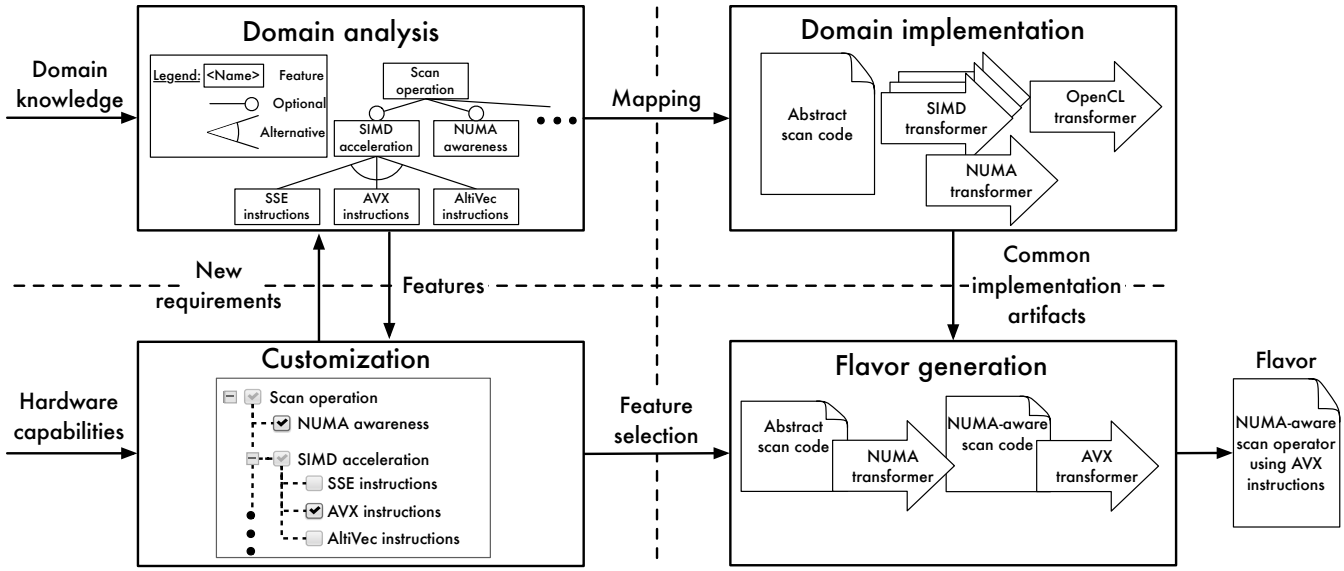

Figure 2: Feature-oriented software-product-line generation.

of different flavors depending on the amount of available processing devices, used compilers, and amount of optimizations. Consequently, the effort to produce optimized code by hand for the given execution environment is immense, which leads to the following challenges.

4.2 Challenges

A simple solution for hardware-tailored database operations is to implement one specialized algorithm for each processing device. To this end, current research focuses on hand-tuned algorithms to one processing device. However, this introduces high implementation and maintenance cost because of the high variability. Consequently, we need an implementation technique fulfilling the following points:

Maximized performance: Produced algorithms should exploit available processor capabilities to achieve the best performance for database operations.

Reduced implementation and maintenance effort: Producing one flavor for each algorithm and processing device per hand incurs much implementation effort, which has to be reduced [14]. Furthermore, flavors of one algorithm tend to have similar code parts, because they produce a similar behavior. Instead of editing each flavor on its own, similar code snippets should be handled together when errors have to be corrected.

Current solutions rely on the usage of OpenCL to produce optimized code for different processing devices [14]. Nevertheless, because of the used abstraction, OpenCL algorithms typically cannot compete with hand-tuned algorithms regarding performance [14]. A promising alternative is used in OmniDB [23], where adapters are used to adapt the code to the underlying hardware. On the one hand, they provide optimized performance on different processing device. However, on the other hand, they have to write an adapter for a new device and have to perform code optimization manually implying high implementation and maintenance effort.

4.3 Possible Solution

To substantiate our vision, we discuss feature-oriented software development as one approach to create database operation algorithms with maximized performance on the specific hardware from a common code base implying reduced implementation and maintenance effort. For this, we propose a reasonable workflow, which generalizes necessary steps for generating hardware-tailored algorithms.

4.3.1 Feature-Oriented Software Development

We propose to use a programming paradigm arising from the software-engineering community to generate tailored code from a common code base. This programming paradigm has been introduced as *feature-oriented software development* (FOSD) and has already been used in a variety of scenarios to construct *software product lines (SPLs)* [3]. SPLs are initially proposed to satisfy requirements of different customers for one product family in order to manage high variability requirements. For this, *features* (which represent an increment in program functionality – e.g., SIMD acceleration) are derived that can be combined to form an end product. As a consequence, different products can be created from a common code base under consideration of different features (e.g., creating a NUMA-aware SIMD-accelerated scan, or a loop-unrolled, parallelized scan). FOSD implies several steps to implement software product lines, as we depict in Figure 2. These steps are *domain analysis*, *domain implementation*, *customization*, and *flavor generation*.

Domain Analysis.

As an initial step, the domain is analyzed to identify possible variability in the product line. As a result, a feature model is constructed which reveals variable implementation parts [3]. In Figure 2, we show an exemplary segment of a feature model for hardware-tailored database operations, which contains instruction sets of different CPUs to enable data parallelism and NUMA awareness. Since we assume that one machine uses only one CPU model, we visualize different instruction sets as alternatives. Furthermore, `SIMD`

```
1  void scan_less(int* array, size_t array_size, int comp_val)
2  {...
3    for(int i=0; i < array_size; ++i){
4      int value = array[i];
5      if(value < comp_val){
6        ...
7      }
8    }
9    ...
10 }
```

Listing 1: Simple scan implementation.

```
1  void scan_less(int* array, size_t array_size, int comp_val)
2  {...
3    //bind thread to the processor holding the data
4    bind_to_proc(getLocalProcessor(array));
5    for(int i=0; i < array_size; ++i){
6      int value = array[i];
7      if(value < comp_val){
8        ...
9      }
10   }
11   ...
12 }
```

Listing 2: NUMA-aware scan implementation.

```
1  void scan_less(int* array, size_t array_size, int comp_val)
2  {...
3    m128i simd_comp = SIMD_VALUE(comp_val);
4    for(int i=0; i < array_size; i+=sizeof(m128i) ){
5      m128i value = SIMD_READ(array[i]);
6      m128 bitmask = SIMD_COMP(value,<,simd_comp);
7      ...
8    }
9    ...
10 }
```

Listing 3: SIMD-accelerated scan implementation.

```
1  void scan_less(int* array, size_t array_size, int comp_val)
2  {...
3    //bind thread to the processor holding the data
4    bind_to_proc(getLocalProcessor(array));
5    m128i simd_comp = SIMD_VALUE(comp_val);
6    for(int i=0; i < array_size; i+=sizeof(m128i) ){
7      m128i value = SIMD_READ(array[i]);
8      m128 bitmask = SIMD_COMP(value,<,simd_comp);
9      ...
10   }
11 ...
12 }
```

Listing 4: NUMA-aware SIMD-accelerated scan.

`acceleration` is optional, because in some scenarios, loop unrolling performs better than SIMD [19]. This step is necessary to structure the variability space of our hardware-tailored database operations and to identify common code of different flavors that has to be implemented in one programming entity.

Domain Implementation.

After the domain analysis, code artifacts are derived that can be combined to fulfill application requirements. Here, because of the tree structure of the feature model, common code of all features is implemented in the parent feature to reduce code overlaps. Notably, the domain implementation is not restricted to a specific implementation technique. Prominent techniques for database systems are, for example, *preprocessor annotations* and *FeatureC++* [18]. For our example in the next section, we motivate the usage of code transformations to implement necessary variability into code as another possibility to produce different flavors.

Customization.

The third step includes the customization of the resulting flavors, in our case to the given hardware. Here, available features from the feature model are selected for the flavor generation.

Flavor Generation.

In the flavor-generation step, resulting flavors of the algorithms are created by using the features selected in the customization step and the programming artifacts of the domain-implementation step. In Figure 2, a scan operator is constructed that is NUMA aware and uses AVX instructions.

4.3.2 Example: A Software Product Line for Scans

For a better understanding of software product lines, we present an example of implementing the scan operator. Our scan product line includes features for NUMA awareness and SIMD acceleration as optional features as visualized in the domain analysis of Figure 2. As an exemplary domain implementation technique, we use code transformations similar to those that are already established in MonetDB [15] for transforming and optimizing query plans. We, in contrast, use this mechanism for tailoring our database operations to the underlying hardware.

To implement a scan product line, we implement an abstract code artifact for a scan in a domain-specific language (DSL). For each feature, we define transformers that translate DSL statements into executable code including additional functionality. For instance, directly generating the code from the DSL, we will produce a simple scan as listed in Listing 1. Instead, when including the NUMA transformer, our scan supports NUMA, as shown in Listing 2. However, if the machine does not use NUMA capabilities, but has to use SIMD instructions, using the SIMD transformer creates the code in Listing 3. So far, we have implemented one transformer per feature. The benefit w.r.t. maintainability is visible when combining both optional optimizations – namely NUMA awareness and SIMD acceleration. Then, both transformers are applied on the abstract scan code producing an optimized scan (cf. Listing 4) without further implementation effort. Considering a high number of n independent code optimizations (e.g., hand unrolling of loops, code with or without branching), instead of implementing each of the 2^n code artifacts by hand, FOSD allows to implement only n code artifacts – in our case transformers – to generate all different flavors. More severely, if the order of the applied optimizations matters, the number of flavors to be tailored by hand is $n!$.

5. CONCLUSION AND FUTURE WORK

In this work, we argue for tailoring database operations to the underlying hardware. We show that there is a variety of different (co-)processors that have to be taken into account for accelerating database operations. More severely, processing devices differ in their processor capabilities, because they were intended to fulfill different tasks. When

using different processing devices to get the best performance out of the used machine, database algorithms have to be tailored to available processor capabilities to exploit their full potential. As a consequence, our vision is to create hardware-tailored database algorithms that take into account the underlying hardware. To this end, we propose feature-oriented software development as a technique to generate optimized algorithms from a common code base. Our vision is that database algorithms are implemented using feature-oriented-software-development techniques to combine the maximal performance of hand-tuned algorithms with low implementation and maintenance cost.

We identify the following steps to realize our vision:

1. **Feature modeling:** Arising from heterogeneous processor capabilities, promising capabilities have to be identified and structured to develop a comprehensive feature model. This includes fine-grained features that exploit the processor capabilities in Section 3.
2. **Implementation technique:** Although, we present a code-transformation-based approach, there are still further techniques that could be used to implement features in an SPL. Whether the transformator-based approach is the best one to go still needs further research.
3. **Mapping features to code:** Arising from the feature model, suitable code snippets to implement a feature have to be identified.
4. **Feedback loop:** Because of the high amount of possible optimizations, we cannot create one flavor for each combination of optimizations, because of the storage consumption. Rather, we have to identify flavors that do not perform optimal and replace them during runtime in a micro-adaptivity way [19].

Finally, we emphasize that this is a continuous process that has to be refined with the upcoming of new processing devices, instruction sets, or further hardware capabilities.

6. REFERENCES

[1] A. Ailamaki, D. J. Dewitt, M. D. Hill, and D. A. Wood. DBMSs on a modern processor: Where does time go? In *VLDB*, pages 266–277, 1999.

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

[3] S. Apel, D. Batory, C. Kästner, and G. Saake. *Feature-oriented software product lines - Concepts and implementation*. Springer, 2013.

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

[5] P. A. Boncz, S. Manegold, and M. L. Kersten. Optimizing database architecture for the new bottleneck: Memory access. *VLDB*, 9(3):231–246, 1999.

[6] P. A. Boncz, M. Zukowski, and N. Nes. MonetDB/X100: Hyper-pipelining query execution. In *CIDR*, pages 225–237, 2005.

[7] S. Breß. Why it is time for a HyPE: A hybrid query processing engine for efficient GPU coprocessing in DBMSs. *The VLDB PhD Workshop, PVLDB*, 6(12):1398–1403, 2013.

[8] 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.

[9] D. Broneske. On the impact of hardware on relational join processing. Master's thesis, University of Magdeburg, 2013.

[10] R. Fang, B. He, M. Lu, K. Yang, N. K. Govindaraju, Q. Luo, and P. V. Sander. GPUQP: Query co-processing using graphics processors. In *SIGMOD*, pages 1061–1063, 2007.

[11] 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 *ACM Trans. Database Syst.*, volume 34. pp. 21:1–21:39. ACM, 2009.

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

[13] J. He, M. Lu, and B. He. Revisiting co-processing for hash joins on the coupled CPU-GPU architecture. In *CoRR*, pages 1–14, 2013.

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

[15] S. Idreos, F. Groffen, N. Nes, S. Manegold, K. S. Mullender, and M. L. Kersten. MonetDB: Two decades of research in column-oriented database architectures. *IEEE Data Eng. Bull.*, 35(1):40–45, 2012.

[16] T. Kiefer, B. Schlegel, and W. Lehner. Experimental evaluation of NUMA effects on database management systems. In *BTW*, pages 185–204, 2013.

[17] 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. *PVLDB*, 2(2):1378–1389, 2009.

[18] M. Rosenmüller, N. Siegmund, H. Schirmeier, J. Sincero, S. Apel, T. Leich, O. Spinczyk, and G. Saake. FAME-DBMS: Tailor-made data management solutions for embedded systems. In *SETMDM*, pages 1–6, 2008.

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

[20] M. Saecker and V. Markl. Big data analytics on modern hardware architectures: A technology survey. In *LNBIP*, volume 138, pages 125 – 149, 2013.

[21] J. Teubner and L. Woods. *Data processing on FPGAs*. Number 35 in Synthesis Lectures on Data Management. Morgan and Claypool Publishers, 2013.

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

[23] S. Zhang, J. He, B. He, and M. Lu. OmniDB: Towards portable and efficient query processing on parallel CPU/GPU architectures. In *VLDB*, pages 1374–1377, 2013.

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