GPU-Accelerated Database Systems: Survey and Open Challenges

Sebastian Breß[1(✉)], Max Heimel[2], Norbert Siegmund[3], Ladjel Bellatreche[4], and Gunter Saake[1]

[1] University of Magdeburg, Magdeburg, Germany
{sebastian.bress,gunter.saake}@ovgu.de
[2] Technische Universität Berlin, Berlin, Germany
max.heimel@tu-berlin.de
[3] University of Passau, Passau, Germany
siegmunn@fim.uni-passau.de
[4] LIAS/ISAE-ENSMA, Futuroscope, Poitiers, France
bellatreche@ensma.fr

Abstract. The vast amount of processing power and memory bandwidth provided by modern graphics cards make them an interesting platform for data-intensive applications. Unsurprisingly, the database research community identified GPUs as effective co-processors for data processing several years ago. In the past years, there were many approaches to make use of GPUs at different levels of a database system. In this paper, we explore the design space of GPU-accelerated database management systems. Based on this survey, we present key properties, important trade-offs and typical challenges of GPU-aware database architectures, and identify major open challenges. Additionally, we survey existing GPU-accelerated DBMSs and classify their architectural properties. Then, we summarize typical optimizations implemented in GPU-accelerated DBMSs. Finally, we propose a reference architecture, indicating how GPU acceleration can be integrated in existing DBMSs.

Keywords: GPU-accelerated database · Survey · Co-processing · Modern database architecture

1 Introduction

Over the last few years, the traditional performance drivers of modern processors – frequency and parallelism – started to hit physical limits. One reason for this is that modern processors are constrained to a certain amount of power they may consume (i.e., the power wall [12]) and further increasing frequency and parallelism would make them overly power hungry. Therefore, hardware

This paper is a substantially extended version of an earlier work [17].

A. Hameurlain et al. (Eds.): TLDKS XV, LNCS 8920, pp. 1–35, 2014.
DOI: 10.1007/978-3-662-45761-0_1

vendors are forced to create processors that are optimized for a certain application field. These developments result in a highly heterogeneous hardware landscape, which is expected to become even more diverse in the future [12]. In order to keep up with the performance requirements of the modern information society, tommorow's database systems will need to exploit and embrace this increased heterogeneity.

In this article, we take a closer look at how today's database engines manage heterogeneous environments, demonstrated by systems that support *Graphics Processing Units* (GPUs). The GPU is the pioneer of modern co-processors, and – in the last decade – it matured from a highly specialized processing device to a fully programmable, powerful co-processor. This development inspired the database research community to investigate methods for accelerating database systems via GPU co-processing. Several research papers and performance studies demonstrate the potential of this approach [7,21,29,32,48,49] – and the technology has also found its way into commercial products (e.g., Jedox [1] or ParStream [2]).

Using graphics cards to accelerate data processing is tricky and has several pitfalls: First, for effective GPU co-processing, the transfer bottleneck between CPU and GPU has to either be reduced or concealed via clever data placement or caching strategies. Second, when integrating GPU co-processing into a real-world *Database Management System* (DBMS), the challenge arises that DBMS internals – such as data structures, query processing and optimization – are traditionally optimized for CPUs. While there is ongoing research on building GPU-aware database systems [22], no unified GPU-aware DBMS architecture has emerged so far.

In this paper, we want to make the community aware of the lack of a unified GPU-aware architecture and derive – based on a literature survey – a reduced design space for such an architecture. In particular, we make the following contributions:

1. We traverse the design space for a GPU-aware database architecture based on results of prior work.
2. We derive research questions that should be investigated by the community to develop GPU-aware database architectures.

Furthermore, as a substantial extension to a previous version of this paper [17], we conducted an in-depth literature survey of eight GPU-accelerated database management systems to validate and refine our theoretical discussions. This complements our findings in proposing a reference architecture. In detail, we make the following additional contributions:

1. We discuss eight *GPU-accelerated DBMSs* (GDBMSs) to review the state-of-the-art, collect prominent findings, and complement our discussion on a GPU-aware DBMS architecture.
2. We create a classification of required architectural properties of GDBMSs.
3. We summarize optimizations implemented by the surveyed systems and derive a general set of optimizations that a GDBMS should implement.

4. We propose a reference architecture for GDBMSs. This architecture provides insights on how to integrate GPU acceleration in main-memory DBMSs.
5. We identify new open challenges compared to our earlier work [17].

We find that GDBMSs should be in-memory column stores, should use the block-at-a-time processing model and exploit all available processing devices for query processing by using a GPU-aware query optimizer. Thus, main memory DBMSs are similar to GPU-accelerated DBMSs, and most in-memory, column-oriented DBMSs can be extended to efficiently support co-processing on GPUs.

The paper is structured as follows: In Sect. 2, we provide necessary background information about GPUs and discuss related work. We explore the design space for GPU-accelerated DBMSs w.r.t. functional and non-functional properties in Sect. 3. In Sect. 4, we survey a representative set of GPU-accelerated DBMSs, classify their architectural properties, summarize possible optimizations to speed up query processing, and propose a reference architecture for GDBMSs. Finally, we identify open challenges for GDBMSs in Sect. 5 and summarize our findings in Sect. 6.

2 Preliminary Considerations

In this section, we provide a brief overview over the architecture of graphics cards, the applied programming model, and related work.

2.1 Graphics Card Architecture

Figure 1 shows the architecture of a modern computer system with a graphics card. The figure shows the architecture of a graphics card from the *Tesla* architecture of NVIDIA. While specific details might be different for other vendors, the general concepts are found in all modern graphic cards. The graphics card – henceforth also called the *device* – is connected to the *host system* via the *PCIExpress bus*. All data transfer between host and device has to pass through this comparably low-bandwidth bus.

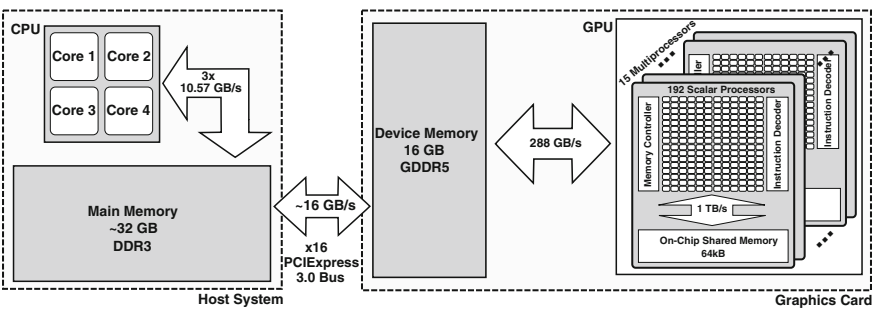

Fig. 1. Overview: Exemplary architecture of a system with a graphics card.

The graphics card itself contains one or more GPUs and a few gigabytes of *device memory*.[1] Typically, host and device do not share the same address space, meaning that neither the GPU can directly access the main memory nor the CPU can directly access the device memory.

The GPU itself consists of a few *multiprocessors*, which can be seen as very wide SIMD processing elements. Each multiprocessor packages several *scalar processors* with a few kilobytes of high-bandwidth, on-chip *shared memory*, cache, and an interface to the device memory.

2.2 Programming a GPU

Programs that run on a graphics card are written in the so-called *kernel programming model*. Programs in this model consist of *host code* and *kernels*. The host code manages the graphics card, initializing data transfer and scheduling program execution on the device. A kernel is a simplistic program that forms the basic unit of parallelism in the kernel programming model. Kernels are scheduled concurrently on several scalar processors in a SIMD fashion: Each kernel invocation - henceforth called *thread* - executes the same code on its own share of the input. All threads that run on the same multiprocessor are logically grouped into a *workgroup*.

One of the most important performance factors in GPU programming is to avoid data transfers between host and device: All data has to pass across the PCIexpress bus, which is the bottleneck of the architecture. Data transfer to the device might therefore consume all time savings from running a problem on the GPU. This becomes especially evident for I/O-bound algorithms: Since accessing the main memory is roughly two to three times faster than sending data across the PCIexpress bus, the CPU will usually have finished execution before the data has even arrived on the device.

Graphics cards achieve high performance through massive parallelism. This means, that a problem should be easy to parallelize to gain most from running on the GPU. Another performance pitfall in GPU programming is caused by divergent code paths. Since each multiprocessor only has a single instruction decoder, all scalar processors execute the same instruction at a time. If some threads in a workgroup diverge, for example due to data-dependent conditionals, the multiprocessor has to serialize the code paths, leading to performance losses. While this problem has been somewhat alleviated in the latest generation of graphics cards, it is still recommended to avoid complex control structures in kernels where possible.

Currently, two major frameworks are used for programming GPUs to accelerate database systems, namely the *Compute Unified Device Architecture* (CUDA) and the *Open Compute Language* (OpenCL). Both frameworks implement the kernel programming model and provide API's that allow the host CPU to manage computations on the GPU and data transfers between CPU and GPU. In contrast to CUDA, which supports NVIDIA GPUs only, OpenCL can run on

[1] Typically around 2–4 GB on mainstream cards and up to 16 GB on high-end devices.

a wide variety of devices from multiple vendors [24]. However, CUDA offers advanced features such as allocation of device memory inside a running kernel or *Uniform Virtual Addressing* (UVA), a technique where CPUs and GPUs share the same virtual address space and the CUDA driver transfers data between CPU and GPU transparently to the application [45].[2]

2.3 Related Work

To the best of our knowledge, there is no survey summarizing the state-of-the-art of GPU-accelerated DBMSs. The only survey we are aware of is from Owens and others, which discusses the state-of-the-art in GPGPU computing [46]. They cover a wide area of research, mainly GPGPU techniques (e.g., stream operations, data structures, and data queries) and GPGPU applications (e.g., databases and data mining, physically-based simulation, and signal and image processing). In contrast to Owens, we focus on recent trends in GPU-accelerated data management to derive a GPU-aware database architecture and open research questions.

3 Exploring the Design Space of a GPU-Aware DBMS Architecture

In this section, we explore the design space of a GPU-accelerated database management system from two points of views: Non-functional properties (e.g., performance and portability) and functional properties (e.g., transaction management and processing model). Note that while we focus on relational systems, most of our discussions apply to other data models as well.

3.1 Non-functional Properties

In the following, we discuss non-functional properties for which DBMSs are typically optimized for, namely performance and portability, and the introduced problems when supporting GPUs. Tsirogiannis and others found that in most cases, the configuration performing best is also the most energy efficient configuration due to the large up-front power consumption in modern servers [59]. Therefore, we will not discuss energy efficiency separately, as energy efficiency is already covered by the performance property.

Performance. Since the GPU is a specialized processor, it is faster on certain tasks (e.g., numerical computations) than CPUs, whereas CPUs outperform GPUs for tasks that are hard to parallelize or that involve complex control flow instructions. He and others observed that joins are 2–7 times faster on the GPU, whereas selections are 2–4 times slower, due to the required data transfers [30].

[2] We are aware that this features are included in OpenCL 2.0 but no OpenCL framework supports this features yet.

The same conclusion was made by Gregg and others, who showed that a GPU algorithm is not necessarily faster than its CPU counterpart, due to the expensive data transfers [27]. One major point for achieving good performance in a GDBMS is therefore to avoid data transfers where possible.

Another problem is how to select the optimal processing device for a given operation. For instance: While the GPU is well suited for easily parallelizable operations (e.g., predicate evaluation, arithmetic operations), the CPU is the vastly better fit when it comes to operations that require complex control structures or significant inter-thread communications (e.g., hash table creation or complex user-defined functions). Selecting the optimal device for a given operation is a non-trivial operation, and – due to the large parameter space (e.g., Breß and others [14] or He and others [29]) – applying simple heuristics is typically insufficient. Breß and others argue that there are four major factors that need to be considered for such a decision (1) the operation to execute, (2) the features of the input data (e.g., data size, data type, operation selectivity, data skew), (3) the computational power and capabilities of the processing devices (e.g., number of cores, memory bandwidth, clock rate), and (4) the load on the processing device (e.g., even if an operation is typically faster on the GPU, one should use the CPU when the GPU is overloaded) [14]. Therefore, we argue that a complex decision model, that incorporates these four factors, is needed to decide on an optimal operator placement.

Portability. Modern DBMSs are tailored towards CPUs and apply traditional compiler techniques to achieve portability across the different CPU architectures (e.g., x86, ARM, Power). By using GPUs – or generally, heterogeneous coprocessors – this picture changes, as CPU code cannot be automatically ported to run efficiently on a GPU. Also, certain GPU toolkits – such as CUDA – bind the DBMS vendor to a certain GPU manufacturer.

Furthermore, processing devices themselves are becoming more and more heterogeneous [55]. In order to achieve optimal performance, each device typically needs its own optimized version of the database operators [19]. However, this means that supporting all combinations of potential devices yields an exponential increase in required code paths, leading to a significant increase in development and maintenance costs.

There are two possibilities to achieve portability also for GPUs: First, we can implement all operators for all vendor-specific toolkits. While this has the advantage that special features of a vendor's product can be used to achieve high performance, it leads to high implementation effort and development costs. Examples for such systems are GPUQP [29] or CoGaDB [13], a column-oriented and GPU-accelerated DBMS. Second, we can implement the operators in a generic framework, such as OpenCL, and let the hardware vendor provide the optimal mapping to the given GPU. While this approach saves implementation effort and simplifies maintenance, it also suffers from performance degradation compared to hand- tuned implementations frameworks. To the best of our knowledge, the only system belonging to the second class is Ocelot [34], which extends MonetDB with OpenCL-based operators.

Summary. From the discussion, it is clearly visible that GPU acceleration complicates the process of optimizing GDBMSs for non-functional properties such as performance and portability. Thus, we need to take special care to achieve comparable applicability with respect to traditional DBMSs.

3.2 Functional Properties

We now discuss the design space for a relational GDBMS with respect to functional properties. We consider the following design decisions: (1) main-memory vs. disk-based system, (2) row-oriented vs. column-oriented storage, (3) processing models (tuple-at-a-time model vs. operator-at-a-time), (4) GPU-only vs. hybrid device database, (5) GPU buffer management (column-wise or page-wise buffer), (6) query optimization for hybrid systems, and (7) consistency and transaction processing (lock-based vs. lock free protocols).

Main-Memory vs. Hard-Disk-Based System. He and others demonstrated that GPU-acceleration cannot achieve significant speedups if the data has to be fetched from disk, because of the IO bottleneck, which dominates execution costs [29]. Since the GPU improves performance only once the data has arrived in main memory, time savings will be small compared to the total query runtime. Hence, a GPU-aware database architecture should make heavy use of in-memory technology.

Row-Stores vs. Column Stores. Ghodsnia compares row and column stores with respect to their suitability for GPU-accelerated query processing [25]. Ghodsnia concluded that a column store is more suitable than a row store, because a column store (1) allows for coalesced memory access on the GPU, (2) achieves higher compression rates (an important property considering the current memory limitations of GPUs), and (3) reduces the volume of data that needs to be transfered. For example, in case of a column store, only those columns needed for data processing have to be transferred between processing devices. In contrast, in a row-store, either the full relation has to be transferred or a projection has to reduce the relation to the data needed to process a query. Both approaches are more expensive than storing the data column wise. Bakkum and others came to the same conclusion [6]. Furthermore, given that we already concluded that a GPU-aware DBMS should be an in-memory database system, and that current research provides an overwhelming evidence in favor of columnar storage for in-memory systems [10]. We therefore conclude that a GPU-aware DBMS should use columnar storage.

Processing Model. There are basically two alternative processing models that are used in modern DBMS: the tuple-at-a-time model [26] and operator-at-a-time bulk processing [42]. Tuple-at-a-time processing has the advantage that intermediate results are very small, but has the disadvantage that it introduces

a higher per tuple processing overhead as well as a high cache miss rate. In contrast, operator-at-a-time processing leads to cache friendly memory access patterns, making effective usage of the memory hierarchy. However, the major drawback is the increased memory requirement, since intermediate results are materialized [42].

Tuple-at-a-time approaches usually apply the so-called *iterator model*, which applies virtual function calls to pass tuples through the required operators [26]. Since graphics cards lack support for virtual function calls – and are notoriously bad at runing the complex control logic that would be necessary to emulate them – this model is unsuited for a GDBMS. Furthermore, we identified in prior work that tuple-wise processing is not possible on the GPU, due to lacking support for inter-kernel communication [15]. We therefore argue that a GDBMS should utilize an operator-at-a-time model.

In order to avoid the IO overhead of this model, multiple authors have suggested a hybrid strategy that uses dynamic code compilation to merge multiple logical operators, or even express the whole query in a single, runtime-generated operator [20,44,60]. Using this strategy, it is not necessary to materialize intermediate results in the GPU's device memory: Tuples are passed between operators in registers, or via shared memory. This approach is therefore an additional potential execution model for a GDBMS.

Database in GPU RAM vs. Hybrid Device Database. Ghodsnia proposed to keep the complete database resident in GPU RAM [25]. This approach has the advantage of vastly reducing data transfers between host and device. Also, since the GPU RAM has a bandwidth that is roughly 16 times higher than the PCIe Bus (3.0), this approach is very likely to significantly increase performance. It also simplifies transaction management, since data does not need to be kept consistent between CPU and GPU.

However, the approach has some obvious shortcomings: First, the GPU RAM (up to ≈16 GB) is rather limited compared to CPU RAM (up to ≈2 TB), meaning that either only small data sets can be processed, or that data must be partitioned across multiple GPUs. Second, a pure GPU database cannot exploit full inter-device parallelism, because the CPU does not any perform data processing. Since CPU and GPU both have their corresponding sweet-spots for different applications (cf. Sect. 3.1), this is a major shortcoming that significantly degrades performance in several scenarios.

Since these problems outweigh the benefits, we conclude that a GDBMS should make use of all available storage and not constrain itself to GPU RAM. While this complicates data processing, and requires a data-placement strategy[3], we still expect the hybrid to be faster than a pure CPU- or GPU-resident system. The performance benefit of using both CPU and GPU for processing was already

[3] Some potential strategies include keeping the hot set of the data resident on the graphics card, or using the limited graphics card memory as a low-resolution data storage to quickly filter out non-matching data items [47].

observed for hybrid query processing approaches (e.g., He and others [29] and Breß and others [18]).

Effective GPU Buffer Management. The buffer-management problem in a CPU/GPU system is similar to the one encountered in traditional disk-based or in-memory systems. That is, we want to process data in a faster, and smaller memory space (GPU RAM), whereas the data is stored in a larger and slower memory space (CPU RAM). The novelty in this problem is, that – in contrast to traditional systems – data can be processed in both memory spaces. In other words: We can transfer data, but we do not have to! This *optionality* opens up some interesting research questions, that have not been covered in traditional database research.

Data structures and data encoding are often highly optimized for the special properties of a processing device to maximize performance. Hence, different kinds of processing devices use an encoding optimized for the respective device. For example, a CPU encoding has to support effective caching to reduce the memory access cost [41], whereas a GPU encoding has to ensure coalesced memory access of threads to achieve maximal performance [45]. This usually requires transcoding data before or after the data transfer, which is an additional overhead that can break performance.

Another interesting design decision is the granularity that should be used for managing the GPU RAM: pages, whole columns, or whole tables? Since we already concluded that a GPU-accelerated database should be columnar, this basically boils down to comparing page-wise vs. column-based caching. Page-wise caching has the advantage that it is an established approach, and is used by almost every DBMS, which eases integration into existing systems. However, a possible disadvantage is that – depending on the page size –, the PCIe bus may be underutilized during data transfers. Since it is more efficient to transfer few large data sets than many little datasets (with the same total data volume) [45], it could be more beneficial to cache and manage whole columns.

Query Placement and Optimization. Given that a GPU-aware DBMS has to manage multiple processing devices, a major problem is to automatically decide which parts of the query should be executed on which device. This decision depends on multiple factors, including the operation, the size & shape of the input data, processing power and computational characteristics of CPU and GPU as well as the optimization criterion. For instance: Optimizing for response time requires to split a query in parts, so that CPU and GPU can process parts of the query in parallel. However, workloads that require a high throughput, need different heuristics. Furthermore, given that we can freely choose between multiple different processing devices with different energy characteristics, non-traditional optimization criteria such as energy-consumption, or cost-per-tuple become interesting in the scope of GPU-aware DBMSs.

He and others were the first to address hybrid CPU/GPU query optimization [29]. They used a Selinger-style optimizer to create initial query plans and

then used heuristics and an analytical cost-model to split a workload between CPU and GPU. In our previous work, we proposed a framework that can perform cost-based operation-wise scheduling and cost-based optimization of hybrid CPU/GPU query plans, which is designed to be used with operator-at-a-time bulk processing [15]. Przymus and others developed a query planner that is capable of optimizing for two goals simultaneously (e.g., query response time and energy consumption) [51]. Heimel and others suggest using GPUs to accelerate query optimization instead of query processing. This approach could help to tackle the additional computational complexity of query optimization in a hybrid system [33]. It should be noted that there is some similarity to the problem of query optimization in the scope of distributed and federated DBMSs [39]. However, there are several characteristics that differentiate distributed from hybrid CPU/GPU query processing:

1. In a distributed system, nodes are autonomous. This is in contrast to hybrid CPU/GPU systems, because the CPU has to explicitly send commands to the co-processors.
2. In a distributed system, there is no global state. By contrast, in hybrid CPU/GPU systems the CPU knows which co-processor performs a certain operation on a specific dataset.
3. The nodes in a distributed system are loosely coupled, meaning that a node may loose network connectivity to the other nodes or might crash. In a hybrid CPU/GPU system, nodes are tightly bound. That is, no network outages are possible due to a high bandwidth bus connection, and a GPU does not go down due to a local software error.

We conclude that traditional approaches for a distributed system do not take into account specifics of hybrid CPU/GPU systems. Therefore, tailor-made co-processing approaches are likely to outperform approaches from distributed or federated query-processing.

Consistency and Transaction Processing. Keeping data consistent in a distributed database is a widely studied problem. But, research on transaction management on the GPU is almost non-existent. The only work we are aware of is by He and others [31] and indicates that a locking-based strategy significantly breaks the performance of GPUs [31]. They developed a lock-free protocol to ensure conflict serializability of parallel transactions on GPUs. However, to the best of our knowledge, there is no work that explicitly addresses transaction management in a GDBMS. It is therefore to be investigated how the performance characteristics of established protocols of distributed systems compare to tailor-made transaction protocols.

Essentially, there are three ways of maintaining consistency between CPU and GPU: (1) Each data item could be kept strictly in one place (e.g., using horizontal or vertical partitioning). In this case, we would not require any replication management and would have to solve a modified allocation problem. (2) We can use established replication mechanisms, such as read one write all or

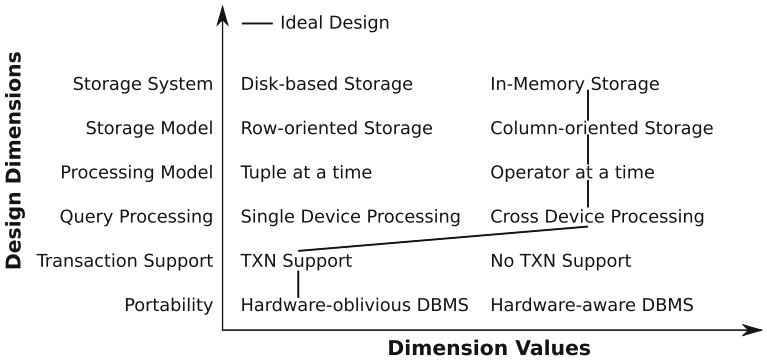

Fig. 2. Design space of GPU-aware DBMSs

primary copy. (3) The system can perform updates always on one processing device (e.g., the CPU) and periodically synchronize these changes to the other devices.

3.3 Summary

We summarize the results of our theoretical discussion in Fig. 2. A GPU-aware database system should reside in-memory and use columnar storage. As processing model, it should implement operator-at-a-time bulk processing model, potentially enhanced by dynamic code compilation. The system should make use of all available (co-)processors in the system (including the CPU!) by having a locality-aware query optimizer, which distributes the workload across all available processing resources. In case the GPU-aware DBMS needs transaction support, it should use an optimistic transaction protocol, such as the timestamp protocol. Finally, in order to reduce implementation overhead, the ideal GDBMS would be hardware-oblivious, meaning all hardware-specific adaption is handled transparently by the system itself.

While this theoretical discussion already gave us a good idea of how the reference architecture for a GDBMS should look like, we will now take a closer look at existing GDBMSs to refine our results.

4 A Survey of GPU-Accelerated DBMSs

In this section, we refine our theoretical discussion of the GDBMS design space from Sect. 3 by conducting a survey on existing GPU-accelerated database systems. First, we describe our research methodology. Second, we discuss the architectural properties of all systems that meet our survey selection criteria. Third, we classify the systems according to our design criteria (cf. Sect. 3). Based on

our classification, we then discuss further optimization techniques used in the surveyed systems. Then, we derive a reference architecture for GPU-accelerated DBMSs based on our results. Finally, we will use this reference architecture for GDBMSs to identify a set of extensions that is required to extend existing main-memory DBMSs to support efficient GPU co-processing.

4.1 Research Methodology

In this section, we state the research questions that drive our survey. Then, we describe the selection criteria to find suitable DBMS architectures in the field of GPU-acceleration. Afterwards, we discuss the properties we focus on in our survey. This properties will be used as base for our classification.

Research Questions

RQ1: Are there recurring architectural properties among the surveyed systems?
RQ2: Are there application-specific classes of architectural properties?
RQ3: Can we infer a reference architecture for GPU-accelerated DBMSs based on existing GPU-accelerated DBMSs?
RQ4: How can we extend existing main-memory DBMSs to efficiently support data processing on GPUs?

Selection Criteria. Since this survey should cover relational GDBMS, we only consider systems that are capable of using the GPU for most relational operations. That is, we disregard stand-alone approaches for accelerating a certain relational operator (e.g., He and others [30,32]), special co-processing techniques (e.g., Pirk and others [49]), or other – non data-processing related – applications for GPUs in database systems [33]. Furthermore, we will not discuss systems using other data models than the relational model, such as graph databases (e.g., Medusa from Zhong and He [64,65]) or MapReduce (e.g., Mars from He and others [28]). Also, given that publications, such as research papers or whitepapers, often lack important architectural informations, we strongly preferred systems that made their source code publicly available. This allowed us to analyze the source code in order to correctly classify the system.

Comparison Properties. According to the design decisions discussed in Sect. 3, we present for each GDBMS the storage system, the storage and processing model, query placement and query optimization, and support for transaction processing. The information for this comparison is taken either directly from analyzing the source code – if available –, or from reading through published articles about the system. If a properties is not applicable for a system, we mark it as not applicable and focus on unique features instead.

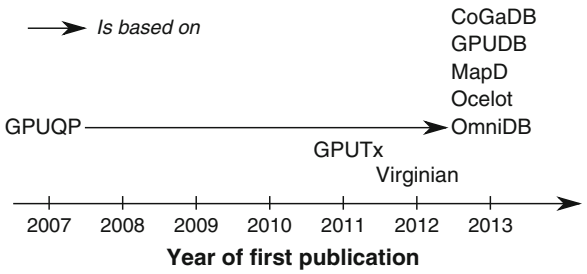

Fig. 3. Time line of surveyed systems.

4.2 GPU-Accelerated DBMS

Based on the discussed selection criteria, we identified the following eight academic[4] systems that are relevant for our survey:

System	Institute	Year	Open Source	Ref.
CoGaDB	University of Magdeburg	2013	yes	[13,18]
GPUDB	Ohio State University	2013	yes	[62]
GPUQP	Hong Kong University of Science and Technology	2007	yes	[29]
GPUTx	Nanyang Technological University	2011	**no**	[31]
MapD	Massachusetts Institute of Technology	2013	**no**	[43]
Ocelot	Technische Universität Berlin	2013	yes	[34]
OmniDB	Nanyang Technological University	2013	yes	[63]
Virginian	NEC Laboratories America	2012	yes	[6]

In Fig. 3, we illustrate the chronological order in which the first publications for each system were published. It is clearly visible that most systems were developed very recently and only few systems are based on older systems. Hence, we expect little influence on the concrete DBMS architecture between each other and hence, a strong external validity of our results.

CoGaDB

Breß and others developed a column-oriented GPU-accelerated DBMS (CoGaDB[5]) [13,18]. CoGaDB focuses on GPU-aware query optimization to achieve efficient co-processor utilization during query processing (Fig. 4).

[4] Note that we deliberately excluded commercial systems such as Jedox [1] or Parstream [2], because they are neither available as open source nor have publications available that provide full architectural details.

[5] Source code available at: http://wwwiti.cs.uni-magdeburg.de/iti_db/research/gpu/cogadb/.

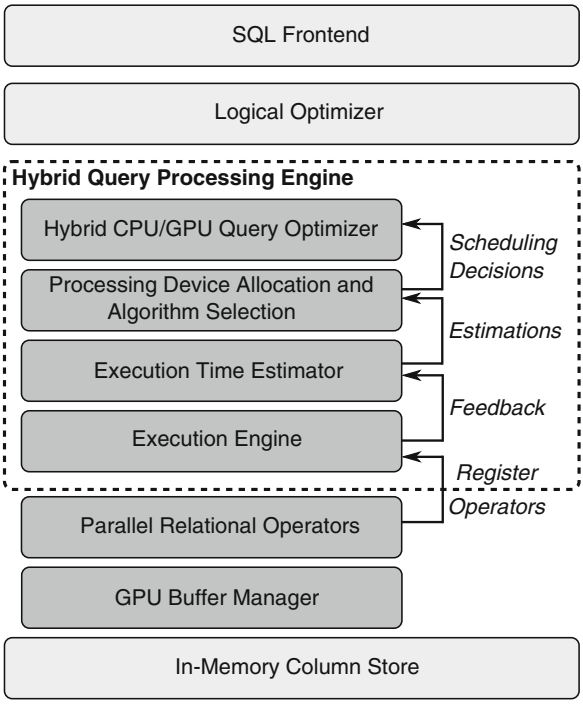

Fig. 4. The architecture of CoGaDB, taken from [16]

Storage System: CoGaDB persists data on disk, but loads the complete database into main memory on startup. If the database is larger than the main memory, CoGaDB relies on the operating system's virtual memory management to swap the least recently used memory pages on disk.

Storage Model: CoGaDB stores data in data structures optimized for in-memory databases. Hence, it stores the data column-wise and compresses VARCHAR columns using dictionary encoding [9]. Furthermore, the data has the same format when stored in the CPU's or the GPU's memory.

Processing Model: CoGaDB uses the operator-at-a-time bulk processing model to make efficient use of the memory hierarchy. This is the basis for efficient query processing using all processing resources.

Query Placement & Optimization: CoGaDB uses the *Hybrid Query Processing Engine* (HyPE) as physical optimizer [13]. HyPE optimizes physical query plans to increase inter-device parallelism by keeping track of the load condition on all (co-)processors (e.g., the CPU or the GPU).

Transactions: Not supported.

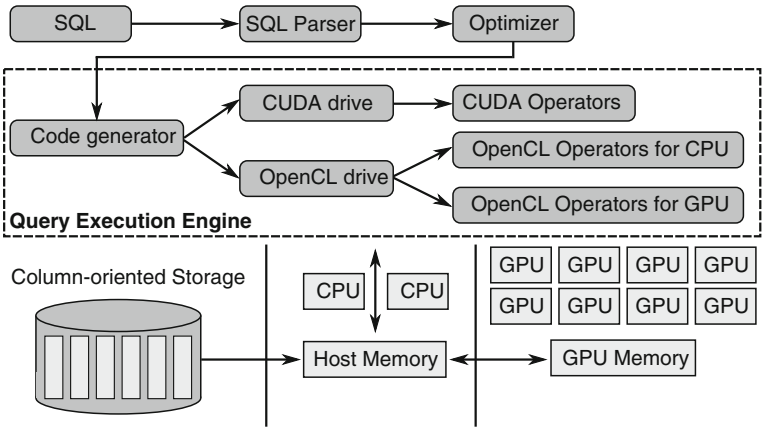

Fig. 5. GPUDB: Query engine architecture, taken from [62]

GPUDB

In order to study the performance behaviour of OLAP queries on GPUs, Yuan and others developed GPUDB[6] [62] (Fig. 5).

Storage System: GPUDB keeps the database in the CPU's main memory to avoid the hard-disk bottleneck. Yuan and others identified a crucial optimization for main-memory DBMS with respect to GPU accelerated execution: In case data is stored in pinned host memory, query execution times can significantly improve (i.e., Yuan and others observed speedups up to 6.5x for certain queries of the *Star Schema Benchmark* (SSB) [52]).

Storage Model: GPUDB stores the data column-wise because GPUDB is optimized for warehousing workloads. Additionally, GPUDB supports common compression techniques (run length encoding, bit encoding, and dictionary encoding) to decrease the impact of the PCIe bottleneck and to accelerate data processing.

Processing Model: GPUDB uses a block-oriented processing model: Blocks are kept in GPU RAM until they are completely processed. This processing model is also known as vectorized processing [54]. Thus, the PCIe bottleneck can be further reduced by overlapping data transfers with computation. For certain queries, Yuan and others observed speedups up to 2.5x compared to no overlapping of processing and data transfers.

GPUDB compiles queries to *driver programs*. A driver program executes a query by calling pre-implemented GPU operators. Hence, GPUDB executes all queries on the GPU and the CPU performs only dispatcher and post processing tasks (i.e., the CPU is used less than 10 % of the time during processing SSB queries [62]).

[6] Source code available at: https://code.google.com/p/gpudb/.

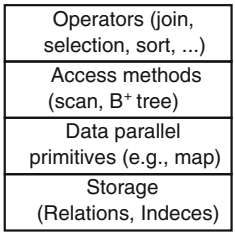

Fig. 6. Execution engine of GPUQP, taken from [29]

Query Placement & Optimization: GPUDB has no support for executing queries on the CPU and GPU in parallel.

Transactions: Not supported.

GPUQP

He and others developed GPUQP[7], a relational query processing system, which stores data in-memory and uses the GPU to accelerate query processing [29]. In GPUQP, each relational operator can be executed on the CPU or the GPU (Fig. 6).

Storage System: GPUQP supports in-memory and disk-based processing. Apparently, GPUQP also attempts to keep data cached in GPU memory. Unfortunately, the authors do not provide any details about the used data placement strategy.

Storage Model: Furthermore, GPUQP makes use of columnar storage and query processing, which fits the hardware capabilities of modern CPUs and GPUs.

Processing Model: GPUQP's basic processing strategy is operator-at-a-time bulk processing. However, GPUQP is also capable of partitioning data for one operator and execute the operator on the CPU and the GPU concurrently. Nevertheless, the impact on the overall performance is small [29].

Query Placement & Optimization: GPUQP combines a Selinger-style optimizer [58] with an analytical cost model to select the cheapest query plan. For each operator, GPUQP allocates either the CPU, the GPU, or both processors (partitioned execution). The query optimizer splits a query plan to multiple sub-plans containing at most ten operators. For each sub-query, all possible plans are created and the cheapest sub-plan is selected. Finally, GPUQP combines the sub-plans to a final physical query plan.

[7] Source code available at: http://www.cse.ust.hk/gpuqp/.

He and others focus on optimizing single queries and do not discuss multi-query optimization. Furthermore, load-aware query scheduling is not considered and there is no discussion of scenarios with multiple GPUs.

Transactions: Not supported.

GPUTx

In order to investigate relational transaction processing on graphics cards, He and others developed GPUTx, a transaction processing engine that runs on the GPU [31].

Storage System & Model: GPUTx keeps all OLTP data inside the GPU's memory to minimize the impact of the PCIe bottleneck. It also applies a columnar data layout to fit the characteristics of modern GPUs.

Processing Model: The processing model is not built on relational operators as in GPUQP. Instead, GPUTx executes pre-compiled stored procedures, which are grouped into one GPU kernel. Incoming transactions are grouped in *bulks*, which are sets of transactions that are executed in parallel on the GPU.

Query Placement & Optimization: Since GPUTx performs the complete data processing on the GPU, query placement approaches are not needed.

Transactions: GPUTx is the only system in our survey – and that we are aware of – that supports running transactions on a GPU. It implements three basic transaction protocols: Two-phase locking, partition-based execution and k-set-based execution. The major finding of GPUTx is that locking-based protocols do not work well on GPUs. Instead, lock-free protocols such as partition-based execution or k-set should be used.

MapD

Mostak develops MapD, which is a data processing and visualization engine, combining traditional query processing capabilities of DBMSs with advanced analytic and visualization functionality [43]. One application scenario is the visualization of twitter messages on a road map[8], in which the geographical position of tweets is shown and visualized as heat map.

Storage System: The data processing component of MapD is a relational DBMS, which can handle data volumes that do not fit the main memory. MapD also tries to keep as much data in-memory as possible to avoid disk accesses.

[8] http://mapd.csail.mit.edu/tweetmap/.

Storage Model: MapD stores data in a columnar layout, and further partitions columns into *chunks*. A chunk is the basic unit of MapD's memory manager. The basic processing model of MapD is processing one operator-at-a-time. Due to the partitioning of data into chunks, it is also possible to process on a per-chunk basis. Hence, MapD is capable of applying block-oriented processing.

Processing Model: MapD processes queries by compiling a query to executable code for the CPU and GPU.

Query Placement & Optimization: The optimizer tries to split a query plan in parts, and processes each part on the most suitable processing device (e.g., text search using an index on the CPU and table scans on the GPU). MapD does not assume that an input data set fits in GPU RAM, and it applies a streaming mechanism for data processing.

Transactions: Not supported.

Ocelot

Heimel and others develop Ocelot[9], which is an OpenCL extension of MonetDB, enabling operator execution on any OpenCL capable device, including CPUs and GPUs [34] (Fig. 7).

Storage System: Ocelot's storage system is built on top of the in-memory model of MonetDB. Input data is automatically transferred from MonetDB to the GPU when needed by an operator. In order to avoid expensive transfers, operator results are typically kept on the GPU. They are only returned at the end of a query, or if the device memory is too filled to fulfill requests. Additionally, Ocelot implements a device cache to keep relevant input data available on the GPU.

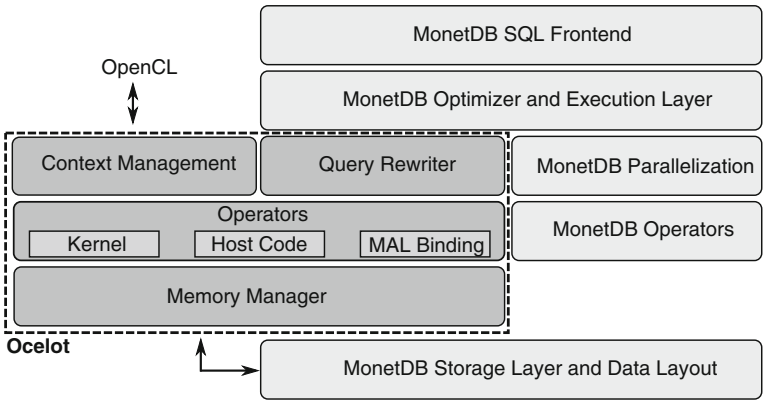

Fig. 7. The architecture of Ocelot, taken from [34]

[9] Source code available at: http://goo.gl/GHeUv.

Storage Model: Ocelot/MonetDB stores data column-wise in *Binary Association Tables* (BATs). Each BAT consists of two columns: One (optional) head storing object identifiers, and one (mandatory) tail storing the actual values.

Processing Model: Ocelot inherits the operator-at-a-time bulk processing model of MonetDB, but extends it by introducing lazy evaluation and making heavy use of the OpenCL event model to forward operator dependency information to the GPU. This allows the OpenCL driver to automatically interleave and reorder operations, e.g., to hide transfer latencies by overlapping the transfer with the execution of a previous operator.

Query Placement & Optimization: In MonetDB, each query plan is represented in the *MonetDB Assembly Language* (MAL) [35]. Ocelot reuses this infrastructures and adds a new query optimizer, which rewrites MAL plans by replacing data processing MAL instructions of vanilla MonetDB with the highly parallel OpenCL MAL instructions of Ocelot.

Query Placement & Optimization: Ocelot does not support cross-device processing, meaning it executes the complete workload either on the CPU or on the GPU.

Transactions: Not supported.

OmniDB

Zhang and others developed OmniDB[10], a GDBMS aiming for good code maintainability while exploiting all hardware resources for query processing [63]. The basic idea is to create a hardware oblivious database kernel (qkernel), which accesses the hardware via *adaptors*. Each adapter implements a common set of operators decoupling the hardware from the database kernel (Fig. 8).

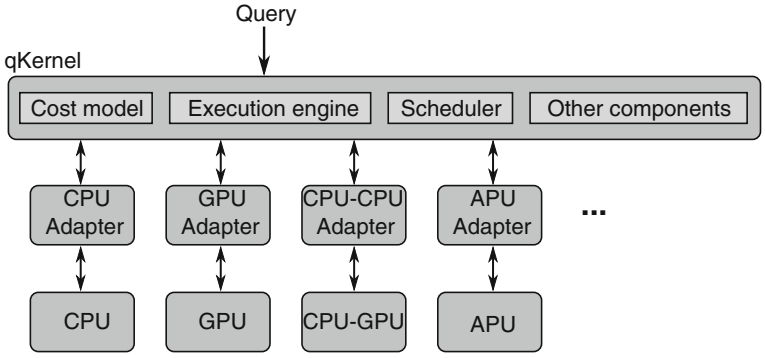

Fig. 8. OmniDB: Kernel adapter design, taken from [63]

[10] Source code available at: https://code.google.com/p/omnidb-paralleldbonapu/.

Storage System & Model: OmniDB is based on GPUQP, and hence, has similar architectural properties to GPUQP. OmniDB keeps data in-memory in a column-oriented data layout.

Processing Model: OmniDB schedules and processes *work units*, which can vary in granularity (e.g., a work unit can be a query, an operator, or a chunk of tuples). Although it is not explicitly mentioned in the paper [63], the fact that OmniDB can process also chunks of tuples is a strong indicator that it supports block-oriented processing.

Query Placement & Optimization: Regarding query placement and optimization, OmniDB chooses the processing device with highest throughput for a *work unit*. To avoid overloading a single device, OmniDB's scheduler ensures that the workload on one processing device may not exceed a certain percentage of the average workload on all processing devices. The cost model relies on the adapters to provide cost functions for the underlying processing devices.

Transactions: Not supported.

Virginian

Bakkum and others develop Virginian[11], which is a GPU-accelerated DBMS keeping data in main memory and supporting filter and aggregation operations on all processing devices [6].

Storage System: Virginian uses no traditional caching of operators, but *uniform virtual addressing* (UVA). This technique allows a GPU kernel to directly access data stored in pinned host memory. The accessed data is transferred over the bus transparently to the device and efficiently overlaps computation and data transfers.

Storage Model: Virgnian implements a data structure called *tablet*, which stores fixed size values column oriented. Additionally, tables can handle variable sized data types such as strings, which are stored in a dedicated section inside the tablet. Thus, Virginian supports strings on the GPU. This is a major difference to other GDBMSs, which apply dictionary compression on strings first and work only on compressed values in the GPU RAM.

Processing Model: Virginian uses operator-at-a-time processing as basic query-processing model. It implements an alternative processing scheme. While most systems call a sequence of highly parallel primitives requiring one new kernel invocation per primitive, Virginian uses the opcode model, which combines all primitives in a single kernel. This avoids writing data back to global memory and reading it again in the next kernel ultimately resulting in block-wise processing on the GPU.

[11] Source code available at: https://github.com/bakks/virginian.

Query Placement & Optimization: Virginian can either process queries on the CPU or on the GPU. Thus, there is no mechanism splitting up the workload between CPU and GPU processing devices and hence, no hybrid query optimizer is available.

Transactions: Not supported.

4.3 Classification

We now classify the surveyed systems according to the architectural properties discussed in Sect. 3.

Storage System: For all eight systems, it is clearly visible that they are designed with main-memory databases in mind, keeping a large fraction of the database in the CPU's main memory (Table 1). GPUQP and MapD also support disk-based data. However, since fetching data from disk is very expensive compared to transferring data over the PCIe bus [29], MapD and GPUQP also keep as much data as possible in main memory. Therefore, we mark all systems as main-memory storage and GPUQP and MapD additionally as disk-based storage.

Storage Model: All systems store their data in a columnar layout, there is no system using row-oriented storage (Table 1). One exception is Virginian, which stores data mainly column-oriented, but also kepps complete rows inside a tablet data structure. This representation is similar to PAX, which stores rows on one page, but stores all records column-wise inside a page [3].

Processing Model: The processing model varies between the surveyed systems (Table 2). The first observation is that no system uses a traditional tuple-at-a-time volcano model [26], as was hypothesized in Sect. 3. Most systems support

Table 1. Classification of Storage System and Storage Model – *Legend:* ✓ – Supported, × – Not Supported, ∘ – Not Applicable

DBMS	Storage System		Storage Model	
	Main-Memory Storage	Disk-based Storage	Column Store	Row Store
CoGaDB	✓	×	✓	×
GPUDB	✓	×	✓	×
GPUQP	✓	✓	✓	×
GPUTx	✓	×	✓	×
MapD	✓	✓	✓	×
Ocelot	✓	×	✓	×
OmniDB	✓	×	✓	×
Virginian	✓	×	✓	×

operator-at-a-time bulk processing [42]. The only exception is GPUTx, which does not support OLAP workloads, because it is an optimized OLTP engine. Hence, we mark the processing model for GPUTx as not applicable. GPUDB, MapD, OmniDB, and Virginian have basic capabilities for block-oriented processing. Additionally, GPUDB and MapD apply a compilation-based query processing strategy.[12] Virginian does not support query compilation. Instead, it uses a single GPU kernel that implements a virtual machine, which calls other GPU kernels (the primitives) in the context of the same kernel, efficiently saving the overhead of reading and writing the result from the GPU's main memory.

Query Placement and Optimization: We identify two major groups of systems: The first group performs nearly all data processing on one processing device (GPUDB, GPUTx, Ocelot, Virginian), whereas the second group is capable of splitting the workload in parts, which are then processed in parallel on the CPU and the GPU (CoGaDB, GPUQP, MapD, OmniDB) (Table 3). We mark systems in the first group as systems that support only single-device processing (SDP), whereas systems of the second group are capable of using multiple devices and thereby allowing cross-device processing (CDP). Note that a system supporting CDP is also capable of executing the complete workload on one processing device (SDP). The hybrid query optimization approaches of CoGaDB, GPUQP, MapD, and OmniDB are mostly greedy strategies or other simple heuristics. It is still an open question how to efficiently trade off between inter-processor parallelization and costly data transfers to achieve optimal performance. For instance: So far, there are no query optimization approaches for machines having multiple GPUs.

Table 2. Classification of Processing Model – *Legend:* ✓ – Supported, × – Not Supported, ○ – Not Applicable

DBMS	Processing Model		
	Operator-at-a-Time	Block-at-a-Time	Just-in-Time Compilation
CoGaDB	✓	×	×
GPUDB	✓	✓	✓
GPUQP	✓	×	×
GPUTx	○	○	○
MapD	✓	✓	✓
Ocelot	✓	×	×
OmniDB	✓	✓	×
Virginian	✓	✓	×

[12] Note that both systems still apply a block-oriented processing model. This is due to the nature of compilation-based strategies, as discussed in Sect. 3.

Table 3. Classification of Query Processing – *Legend:* ✓ – Supported, × – Not Supported, ∘ – Not Applicable

DBMS	Query Processing	
	Single-Device Processing	Cross-Device Processing
CoGaDB	✓	✓
GPUDB	✓	×
GPUQP	✓	✓
GPUTx	✓	×
MapD	✓	✓
Ocelot	✓	×
OmniDB	✓	✓
Virginian	✓	×

Transaction Processing: Apart from GPUTx, none of the surveyed GDBMSs support transactions (Table 4). GPUTx keeps data strictly in the GPU's RAM, and needs to transfer only incoming transactions to the GPU and the result back to the CPU. Since GPUTx achieved a 4–10 times higher throughput than a comparable CPU-based OLTP engine, there is a need for further research in the area of transaction processing in GDBMSs so that OLTP systems can also benefit from GPU acceleration. Apparently, online analytical processing and online transactional processing can be significantly accelerated by using GPU acceleration. However, it is not yet clear which workload type is more suitable for which processing device type. Furthermore, the efficient combination of OLTP/OLAP workloads is still an active research field (e.g., Kemper and Neumann [38]). Thus, it is an open question whether and under which circumstances GPU-acceleration is beneficial for such hybrid OLTP/OLAP workloads.

Portability: The only GDBMSs having a portable, hardware-oblivious database architecture are Ocelot and OmniDB. All other systems are either tailored to a vendor specific programming framework or have no technique to hide the details of the device-specific operators in the architecture. Ocelot's approach has the advantage that only a single set of parallel database operators has to be implemented, which can then be mapped to all processing devices supporting OpenCL (e.g., CPUs, GPUs, or Xeon Phis). By contrast, OmniDB uses an adapter interface, in which each adapter provides a set of operators and cost functions for a certain processing-device type. It is unclear, which approach will lead to the best performance/maintainability ratio, and how large the performance loss is compared to a hardware-aware system. However, if portability can be achieved with only a small performance degradation, it would substantially benefit the maintainability and applicability of GDBMSs [63]. Hence, the trend towards hardware-oblivious DBMSs is likely to continue.

Table 4. Classification of Transaction Support and Portability – *Legend:* ✓ – Supported, × – Not Supported, ∘ – Not Applicable

DBMS	Transaction Support	Portability	
		Hardware Aware	Hardware Oblivious
CoGaDB	×	✓	×
GPUDB	×	✓	×
GPUQP	×	✓	×
GPUTx	✓	✓	×
MapD	×	✓	×
Ocelot	×	×	✓
OmniDB	×	×	✓
Virginian	×	✓	×

4.4 Potential Optimizations for GDBMSs

We will now discuss and summarize potential optimizations, which a GDBMS may implement to make full use of the underlying hardware in a hybrid CPU/GPU system. Additionally, we briefly discuss existing approaches for each optimization. As already discussed, data transfers have the highest impact on GDBMS performance. Hence, every optimization avoiding or minimizing the impact of data transfers are mandatory. We refer to these optimizations as cross-device optimizations. Based on our surveyed systems, we could identify the following *cross-device optimizations*:

Efficient Data Placement Strategy: There are two possibilities to manage the GPU RAM. The first possibility is an explicit management of data on GPUs using a buffer-management algorithm. The second possibility is using mechanisms such as *Unified Virtual Addressing* (UVA), which enables a GPU kernel to directly access the main memory. Kaldewey and others observed a significant performance gain (3-8x) using UVA for Hash Joins on the GPU compared to the CPU [37]. Furthermore, data has not to be kept consistent between CPU and GPU, because there is no "real" copy in the GPU RAM. However, this advantage can also be a disadvantage, because caching data in the GPU RAM can avoid the data transfer from the CPU to the GPU.

GPU-aware Query Optimizer: A GDBMS should make use of all processing devices to maximize performance. Therefore, it should offload operations to the GPU. However, offloading single operations of a query plan does not necessarily accelerate performance. Hence, a GPU-aware optimizer has to identify sub plans of a query plan, which it can process on the CPU or the GPU [29]. Furthermore, the resulting plan should minimize the number of copy operations [15]. Since optimizers are typically cost based, a GDBMS needs for each GPU operator a cost model. The most common approach is to use analytical models (e.g., Manegold and others for the CPU [40] and He and

others for the GPU [29]). However, with the increasing hardware complexity, machine-learning-based models become increasingly popular [14].

Data Compression: The data placement and query optimization techniques attempt to avoid data transfers as much as possible. To reduce overhead in case a GDBMS has to perform data transfers, the data volume can be reduced by compression techniques. Thus, compression can significantly decrease processing costs [62]. Fang and others discussed an approach, which combines different lightweight compression techniques to compress data at the GPU [23]. They developed a planner for cascading compression techniques, which decides on a suitable subset and order of available compression techniques. Przymus and Kaczmarski focused on compression for time-series databases on the GPU [50]. Andrzejewski and Wrembel discussed compression of bitmap indexes on the GPU [4].

Overlap of Data Transfer and Processing: The second way to accelerate processing, in case a data transfer needs to performed, is overlapping the execution of a GPU operator with a data transfer operation [6,62]. This optimization keeps all hardware components busy, and basically narrows down the performance of the system to the PCIe bus bandwidth.

Pinned Host Memory: The third way to accelerate query processing in case we have to perform a copy operation is keeping data in pinned host memory. This optimization saves one indirection, because the DMA controller can transmit data directly to the device [62]. Otherwise, data has to be copied in pinned memory first, introducing additional latency in data transmission. However, using pinned host memory has the drawback that the amount of available pinned host memory is much smaller than the amount of unpinned memory (i.e., memory that can be paged to disk by the virtual memory manager) [56]. Therefore, a GDBMS has to decide which data it should keep in pinned host memory. It is still an open issue how much memory should be spent on a pinned host memory buffer for faster data transfers to the GPU.

Figure 9 illustrates the identified cross-device optimizations and the relationships between them.

The second class of optimizations we identified, targets the efficiency of operator execution on a single processing device. We refer to this class of optimizations as *device-dependent optimizations*. Since we focus on GPU-aware systems, we only discuss optimizations for GPUs. Based on the surveyed systems, we summarize the following GPU-dependent optimizations:

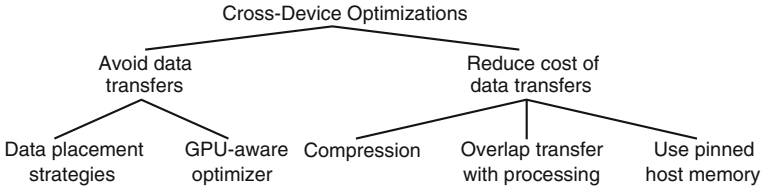

Fig. 9. Cross-device optimizations

Block-oriented Query Processing: A GDBMS can avoid the overhead of writing results of an operator back to a processing device's main memory by processing data on a per block basis rather than on a per operator basis. The idea is to process data already stored in the cache (CPU) or shared memory (GPU), which saves memory bandwidth and significantly increases performance of query processing [11,62]. Additionally, block-oriented processing is a necessary prerequisite for overlapping processing and data transfer for single operations and allows for a more fine grained workload distribution on available processing devices [63]. Note that traditional pipelining of blocks between GPU operators is not possible, because inter-kernel communication is undefined [15]. While launching a new kernel for each block is likely to be expensive, query compilation and kernel fusion are promising ways to allow block-oriented processing on the GPU as well.

Compilation-based Query Processing: Compiling queries to executable code is a common optimization in main-memory DBMSs [20,44,60]. As already discussed, query compilation allows for block-oriented processing on GPUs as well and achieves a significant speedup compared to primitive-based query processing (e.g., operator-at-a-time processing [29]). However, query compilation introduces additional overhead, because compiling a query to executable code typically is more expensive than building a physical query execution plan. Yuan and others overcome this shortcoming by pre-compiling operators. Thus, they only need to compile the query plan itself to a driver program [62]. A similar approach called *kernel weaver* is used by Wu and others [61]. They combine CUDA kernels for relational primitives into one kernel. This has the advantage that the optimization scope is larger and the compiler can perform more optimizations. However, the disadvantage is the increased compilation time. Rauhe and others introduce in their approach two processing phases: compute and accumulate. In the compute phase, a number of threads are assigned to a partition of the input data and each thread performs all operations of a query on one tuple and then, continues with the next tuple, until the thread processed its partition. In the accumulate phase, the intermediate results are combined to the final result [53].

All-in-one Kernel: A promising alternative to compilation-based approaches is to combine all relational primitives in one kernel [6]. Thus, a relational query has to be translated to a sequence of op codes. An op code identifies the next primitive to be executed. Therefore, it is basically an on-GPU virtual machine, which saves the initial overhead of query compilation. However, the drawback is a limited optimization scope compared to kernel weaver [61].

Portability: Until now, we mainly discussed performance optimizations. However, each of the discussed optimizations are mainly implemented device dependent. This increases the overall complexity of a GDBMS. The problem gets even more complex with new processing device types such as accelerated processing units or the Intel Xeon Phi. Heimel and others implemented a hardware oblivious DBMS kernel in OpenCL and still achieved a significant acceleration of query processing [34]. Zhang and others implemented *q-kernel*, a hardware-oblivious database kernel using device adapters to the

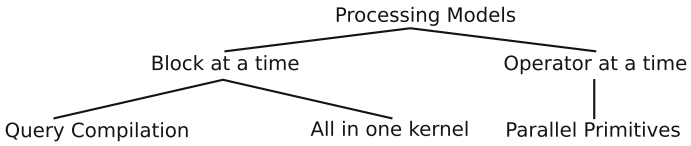

Fig. 10. Device-dependent optimizations: Efficient processing models

underlying processing devices [63]. It is still not clear which part of a kernel should be hardware oblivious and which part should be hardware aware. For the parts that have to be hardware aware, modern software engineering methods such as software product lines can be used to manage the GDBMS's complexity [19].

Figure 10 illustrates the identified device-dependent optimizations and the relationships between them.

4.5 A Reference Architecture for GPU-Accelerated DBMSs

Based on our in-depth survey of existing GDBMSs, we now derive a reference architecture for GDBMSs. After careful consideration of all surveyed systems, we decided to use the GPUQP [29]/OmniDB [63] architecture as basis for our reference architecture, because they already include a major part of the common properties of the surveyed systems. We illustrate the reference architecture in Fig. 11.

We will describe the query-evaluation process in a top-down view. On the upper levels of the query stack, a GPU-accelerated DMBS is virtually identical to a "traditional" DBMS. It includes functionality for integrity control, parsing SQL queries, and performing logical optimizations on queries. Major differences between main-memory DBMSs and GDBMSs emerge in the physical optimizer. While classical systems choose the most suitable access structure and algorithm to operate on the access structure, a GPU-accelerated DBMS has to additionally decide for each operator on a processing device. For this task, a GDBMS needs refined[13] cost models that also predict the cost for GPU and CPU operations. Based on these estimates, a scheduler can allocate the cheapest processing device. Furthermore, a query should make use of multiple processing devices to speed up execution. Hence, the physical optimizer has to optimize hybrid CPU/GPU query plans, which significantly increases the optimization space.

Relational operations are implemented in the next layer. These operators typically use access structures to process data. In GDBMSs, access structures have to be reimplemented on GPUs to achieve a high efficiency. However, depending

[13] Since these models need to be able to estimate comparable operator runtimes across different devices, we and others [13] argue that dynamic cost models, which apply techniques from Machine Learning to adapt to the current hardware, are likely required here.

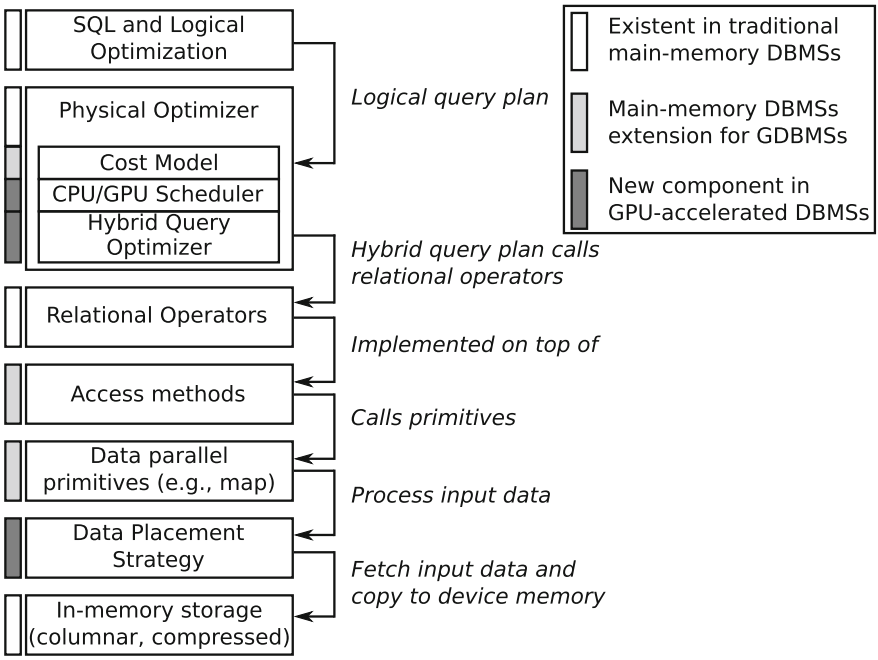

Fig. 11. Layered architecture of GDBMSs

on the processing device chosen by the CPU/GPU scheduler, different access structures are available. This is an additional dependency the query optimizer needs to take into account.

Then, a set of parallel primitives can be applied to an access structure to process a query. In this component, the massive parallelism of CPUs and GPUs is fully used to speed up query processing. However, a GPU operator can only work on data stored in GPU memory. Hence, all access structures are built on top of a data-placement component, that caches data on a certain processing device, depending on the access patterns of the workload (e.g., certain columns for column scans or certain nodes of tree indexes [8,57]). Note that the data-placement strategy is the most performance critical component in a GDBMS due to the major performance impact of data transfers.

The backbone of a GDBMS is a typical in-memory storage, which frequently stores data in a column-oriented format.[14] Compression techniques are not only beneficial in keeping the major part of a database in-memory, compression also reduces the impact of the PCIe bottleneck.

[14] We are aware that some in-memory DBMSs can also store data row-oriented, such as HyPer [38]. However, in GDBMSs, row-oriented storage either increases the data volume to be transfered or requires a projection operation before the transfer. A row-oriented layout also makes it difficult to achieve optimal memory access patterns on a GPU.

4.6 Summary: Extension Points for Main-Memory DBMSs

In summary, we can extend most main-memory DBMSs supporting column-oriented data layout and bulk processing to be GPU-accelerated DBMSs. We identify the following extension points: Cost models, CPU/GPU scheduler, hybrid query optimizer, access structures and algorithms for the GPU, and a data placement strategy.

Cost Models: For each processor, we need to estimate the execution time of an operator. This can be either done by analytical cost models (e.g., Manegold and others for CPUs [40] and He and others for GPUs [29]) or learning-based approaches (e.g., Breß and others [14] or Ilić and Sousa [36]).

CPU/GPU Scheduler: Based on the cost models, a scheduler needs to allocate processing devices for a set of operators (e.g., CHPS from Ilić and Sousa, HyPE from Breß and others [14], or StarPU from Augonnet and others [5]).

Hybrid Query Optimizer: The query optimizer needs to consider the data transfer bottleneck and memory requirements of operators to create a suitable physical execution plan. Thus, the optimizer should make use of cost models, a CPU/GPU scheduler, and heuristics minimizing the time penalty of data transfers (e.g., HyPE from Breß and others [14]).

Access structures and algorithms for the GPU: In order to support GPU-acceleration, a DBMS needs to implement access structures on the GPU (e.g., columns or B^+-trees) and operators that work on them. Here, the most approaches were developed [7,21,29,32,48,49].

Data Placement Strategy: A DBMS needs to keep track of which data is stored on the GPU, and which access structure needs to be transferred to GPU memory [29]. Aside from a manual memory management, it is also possible to use techniques such as UVA and let the GPU driver handle the data transfers transparently to the DBMS [62]. However, this may result in less efficiency because a manual memory management can exploit knowledge about the DBMS and the workload.

Implementing these extensions is a necessary precondition for a DBMS to support GPU co-processing efficiently.

5 Open Challenges and Research Questions

In this section, we identify *open challenges* for GPU-accelerated DBMSs. We differentiate between two major classes of challenges, namely the IO bottleneck, which includes disk IO as well as data transfers between CPU and GPU, and query optimization.

5.1 IO Bottleneck

In a GDBMS, there are two major IO bottlenecks. The first is the classical disk IO, and the second bottleneck is the PCIe bus. As for the latter bottleneck, we can differ between avoiding and reducing the impact of the bottleneck.

Disk-IO Bottleneck: GPU-accelerated operators are of little use for disk-based database systems, where most time is spent on disk I/O. Since the GPU improves performance only once the data is in main memory, time savings will be small compared to the total query runtime. Furthermore, disk-resident databases are typically very large, making it harder to find an optimal data placement strategy. However, database systems can benefit from GPUs even in scenarios where not the complete database fits into main memory. As long as the *hot data* fits into main memory, GPUs can accelerate data processing. It is still an open problem to which degree a database has to fit into the CPU's main memory, so GPU acceleration pays off.

Data Placement Strategy: GPU-accelerated databases try to keep relational data cached on the device to avoid data transfer. Since device memory is limited, this is often only possible for a subset of the data. Deciding which part of the data should be offloaded to the GPU – finding a *data placement strategy* – is a difficult problem that currently remains unsolved.

Reducing PCIe Bus Bottleneck: Data transfers can be significantly accelerated by keeping data in pinned host memory. However, the amount of available pinned memory is much more limited compared to the amount of available virtual memory. Therefore, a GDBMS has to decide which data to keep in pinned memory. Since data is typically cached in GPU memory, a GDBMS needs a multi-level caching technique, which is yet to be found.

5.2 Query Optimization

In GDBMSs, query processing and optimization have to cope with new challenges. We identify as major open challenges a generic cost model, an increased complexity of query optimization due to the larger optimization space, insufficient support for using multi-processing devices for query-compilation approaches, and accelerating different workload types.

Generic Cost Model: From the query-optimization perspective, a GDBMS needs a cost model to perform cost-based optimization. In this area, two basic cost-model classes have emerged. The first class consists of analytical cost models and the second class makes use of machine-learning approaches to learn cost models for some training data. While analytical cost models excel in computational efficiency, learning-based strategies need no knowledge about the underlying hardware and can adapt to changing data. It is still open which kind of cost model is optimal for GDBMSs.

Increased Complexity of Query Optimization: Having the option of running operations on a GPU increases the complexity of query optimization: The plan search space is significantly larger and a cost function that compares run-times across architectures is required. While there has been prior work in this direction [14,15,29], GPU-aware query optimization remains an open challenge.

Query Compilation for Multiple Devices: With the upcoming trend of query compilation, the basic problem of processing-device allocation remains

the same as in traditional query optimization. However, as of now, the available compilation approaches only compile complete queries for either the CPU or the GPU. It is still an open challenge how to compile queries to code that uses more than one processing device concurrently.

Considering different Workload Types: OLTP as well as OLAP workloads can be significantly accelerated using GPUs. Furthermore, it became common to have a mix of both workload types in a single system. It remains open, which workload types are more suited for which processing-device type and how to efficiently schedule OLTP and OLAP queries on the CPU and the GPU.

6 Conclusion and Future Directions

The performance of modern processors is no longer bound primarily by transistor density but by a fixed energy budget, the *power wall* [12]. Whereas CPUs often spend additional chip space on more cache capacity, other processors spend most of their chip space on light-weight cores, which omit heavy control logic and are thus, more energy efficient. Therefore, future machines will likely consist of a set of heterogeneous processors, having CPUs and specialized co-processors such as GPUs, Multiple Integrated Cores (MICs), or FPGAs. Hence, the question of using co-processors in databases is not *why* but *how* we can do this most efficiently.

The pioneer of modern co-processors is the GPU, and many prototypes of GPU-accelerated DBMSs have emerged over the past seven years implementing new co-processing approaches and proposing new system architectures. We argue that we need to take into account tomorrows hardware in today's design decisions. Therefore, in this paper, we theoretically explored the design space of GPU-aware database systems. In summary, we argue that a GDBMS should be an in-memory, column-oriented DBMS using the block-at-a-time processing model, possibly extended by a just-in-time-compilation component. The system should have a query optimizer that is aware of co-processors and data-locality, and is able to distribute a workload across all available (co-)processors.

We validated these findings by surveying the implementation details of eight existing GDBMSs and classifying them along the mentioned dimensions. Additionally, we summarized common optimizations implemented in GDBMSs and inferred a reference architecture for GDBMSs, which may act as a starting point in integrating GPU-acceleration in popular main-memory DBMSs. Finally, we identified potential *open challenges* for further development of GDBMSs.

Our results are not limited to GPUs, but should also be applicable to other co-processors. The existing techniques can be applied to virtually all massively parallel processors having dedicated high-bandwidth memory with limited storage capacity.

Acknowledgements. We thank Tobias Lauer from Jedox AG and the anonymous reviewers of the GPUs in Databases Workshop for their helpful feedback on the workshop

version of this paper [17]. We thank Jens Teubner from TU Dortmund University, Michael Saecker from ParStream GmbH, and the anonymous reviewers of the TLDKS journal for their helpful comments on the journal version of this paper.

References

1. Palo GPU accelerator. White Paper (2010)
2. Parstream - turning data into knowledge. White Paper, November 2010
3. Ailamaki, A., DeWitt, D.J., Hill, M.D., Skounakis, M.: Weaving relations for cache performance. In: VLDB, pp. 169–180. Morgan Kaufmann Publishers Inc. (2001)
4. Andrzejewski, W., Wrembel, R.: GPU-WAH: applying GPUs to compressing bitmap indexes with word aligned hybrid. In: Bringas, P.G., Hameurlain, A., Quirchmayr, G. (eds.) DEXA 2010, Part II. LNCS, vol. 6262, pp. 315–329. Springer, Heidelberg (2010)
5. Augonnet, C., Thibault, S., Namyst, R., Wacrenier, P.-A.: StarPU: a unified platform for task scheduling on heterogeneous multicore architectures. Concurr. Comput. Pract. Exp. **23**(2), 187–198 (2011)
6. Bakkum, P., Chakradhar, S.: Efficient data management for GPU databases (2012). http://pbbakkum.com/virginian/paper.pdf
7. Bakkum, P., Skadron, K.: Accelerating SQL database operations on a GPU with CUDA. In: GPGPU, pp. 94–103. ACM (2010)
8. Beier, F., Kilias, T., Sattler, K.-U.: GiST scan acceleration using coprocessors. In: DaMoN, pp. 63–69. ACM (2012)
9. Binnig, C., Hildenbrand, S., Färber, F.: Dictionary-based order-preserving string compression for main memory column stores. In: SIGMOD, pp. 283–296. ACM (2009)
10. Boncz, P.A., Kersten, M.L., Manegold, S.: Breaking the memory wall in MonetDB. Commun. ACM **51**(12), 77–85 (2008)
11. Boncz, P.A., Zukowski, M., Nes, N.: MonetDB/X100: hyper-pipelining query execution. In: CIDR, pp. 225–237 (2005)
12. Borkar, S., Chien, A.A.: The future of microprocessors. Commun. ACM **54**(5), 67–77 (2011)
13. Breß, S.: Why it is time for a HyPE: a hybrid query processing engine for efficient GPU coprocessing in dbms. The VLDB PhD Workshop, PVLDB **6**(12), 1398–1403 (2013)
14. Breß, S., Beier, F., Rauhe, H., Sattler, K.-U., Schallehn, E., Saake, G.: Efficient co-processor utilization in database query processing. Inf. Syst. **38**(8), 1084–1096 (2013)
15. Breß, S., Geist, I., Schallehn, E., Mory, M., Saake, G.: A framework for cost based optimization of hybrid CPU/GPU query plans in database systems. Control Cybern. **41**(4), 715–742 (2012)
16. Breß, S., Haberkorn, R., Ladewig, S.: CoGaDB reference manual (2014). http://wwwiti.cs.uni-magdeburg.de/iti_db/research/gpu/cogadb/0.3/doc/refman.pdf
17. Breß, S., Heimel, M., Siegmund, N., Bellatreche, L., Saake, G.: Exploring the design space of a GPU-aware database architecture. In: Catania, B., et al. (eds.) New Trends in Databases and Information Systems. AISC, vol. 241, pp. 225–234. Springer, Heidelberg (2014)

18. Breß, S., Siegmund, N., Bellatreche, L., Saake, G.: An operator-stream-based scheduling engine for effective GPU coprocessing. In: Catania, B., Guerrini, G., Pokorný, J. (eds.) ADBIS 2013. LNCS, vol. 8133, pp. 288–301. Springer, Heidelberg (2013)
19. Broneske, D., Breß, S., Heimel, M., Saake, G.: Toward hardware-sensitive database operations. In: EDBT, pp. 229–234. OpenProceedings.org (2014)
20. Dees, J., Sanders, P.: Efficient many-core query execution in main memory column-stores. In: ICDE, pp. 350–361. IEEE (2013)
21. Diamos, G., Wu, H., Lele, A., Wang, J., Yalamanchili, S.: Efficient relational algebra algorithms and data structures for GPU. Technical report, Center for Experimental Research in Computer Systems (CERS) (2012)
22. Fang, R., He, B., Lu, M., Yang, K., Govindaraju, N.K., Luo, Q., Sander, P.V.: GPUQP: query co-processing using graphics processors. In: SIGMOD, pp. 1061–1063. ACM (2007)
23. Fang, W., He, B., Luo, Q.: Database compression on graphics processors. PVLDB **3**, 670–680 (2010)
24. Gaster, B.R., Howes, L., Kaeli, D., Mistry, P., Schaa, D.: Heterogeneous Computing With Opencl. Elsevier Sci. Technol. 1–2 (2012)
25. Ghodsnia, P.: An in-GPU-memory column-oriented database for processing analytical workloads. In: The VLDB PhD Workshop. VLDB Endowment (2012)
26. Graefe, G.: Encapsulation of parallelism in the volcano query processing system. In: SIGMOD, pp. 102–111. ACM (1990)
27. Gregg, C., Hazelwood, K.: Where is the data? why you cannot debate CPU vs. GPU performance without the answer. In: ISPASS, pp. 134–144. IEEE (2011)
28. He, B., Fang, W., Luo, Q., Govindaraju, N.K., Wang, T.: Mars: a mapreduce framework on graphics processors. In: PACT, pp. 260–269. ACM (2008)
29. He, B., Lu, M., Yang, K., Fang, R., Govindaraju, N.K., Luo, Q., Sander, P.V.: Relational query co-processing on graphics processors. In: ACM Transactions on Database System, vol. 34. ACM (2009)
30. He, B., Yang, K., Fang, R., Lu, M., Govindaraju, N., Luo, Q., Sander, P.: Relational joins on graphics processors. In: SIGMOD, pp. 511–524. ACM (2008)
31. He, B., Yu, J.X.: High-throughput transaction executions on graphics processors. PVLDB **4**(5), 314–325 (2011)
32. He, J., Lu, M., He, B.: Revisiting co-processing for hash joins on the coupled CPU-GPU architecture. PVLDB **6**(10), 889–900 (2013)
33. Heimel, M., Markl, V.: A first step towards GPU-assisted query optimization. In: ADMS. VLDB Endowment (2012)
34. Heimel, M., Saecker, M., Pirk, H., Manegold, S., Markl, V.: Hardware-oblivious parallelism for in-memory column-stores. PVLDB **6**(9), 709–720 (2013)
35. Idreos, S., Groffen, F., Nes, N., Manegold, S., Mullender, K.S., Kersten, M.L.: MonetDB: Two decades of research in column-oriented database architectures. IEEE Data Eng. Bull. **35**(1), 40–45 (2012)
36. Ilić, A., Sousa, L.: CHPS: an environment for collaborative execution on heterogeneous desktop systems. Int. J. Netw. Comput. **1**(1), 96–113 (2011)
37. Kaldewey, T., Lohman, G., Mueller, R., Volk, P.: GPU join processing revisited. In: DaMoN, pp. 55–62. ACM (2012)
38. Kemper, A., Neumann, T.: HyPer: a hybrid OLTP&OLAP main memory database system based on virtual memory snapshots. In: ICDE, pp. 195–206. IEEE (2011)
39. Kossmann, D.: The state of the art in distributed query processing. ACM Comput. Surv. **32**(4), 422–469 (2000)

40. Manegold, S., Boncz, P., Kersten, M.L.: Generic database cost models for hierarchical memory systems. In: PVLDB, pp. 191–202. VLDB Endowment (2002)
41. Manegold, S., Boncz, P.A., Kersten, M.L.: Optimizing database architecture for the new bottleneck: memory access. VLDB J. **9**(3), 231–246 (2000)
42. Manegold, S., Kersten, M.L., Boncz, P.: Database architecture evolution: mammals flourished long before dinosaurs became extinct. PVLDB **2**(2), 1648–1653 (2009)
43. Mostak, T.: An overview of MapD (massively parallel database). White Paper, Massachusetts Institute of Technology, April 2013. http://geops.csail.mit.edu/docs/mapd_overview.pdf
44. Neumann, T.: Efficiently compiling efficient query plans for modern hardware. PVLDB **4**(9), 539–550 (2011)
45. NVIDIA. NVIDIA CUDA C programming guide, pp. 31–36, 40, 213–216, Version 6.0. (2014). http://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf. Accessed 21 April 2014
46. Owens, J.D., Luebke, D., Govindaraju, N., Harris, M., Krger, J., Lefohn, A.E., Purcell, T.J.: A survey of general-purpose computation on graphics hardware. Comput. Graph. Forum **26**(1), 80–113 (2007)
47. Pirk, H.: Efficient cross-device query processing. In: The VLDB PhD Workshop. VLDB Endowment (2012)
48. Pirk, H., Manegold, S., Kersten, M.: Accelerating foreign-key joins using asymmetric memory channels. In: ADMS, pp. 585–597. VLDB Endowment (2011)
49. Pirk, H., Manegold, S., Kersten, M.: Waste not... efficient co-processing of relational data. In: ICDE. IEEE (2014)
50. Przymus, P., Kaczmarski, K.: Dynamic compression strategy for time series database using GPU. In: Catania, B., et al. (eds.) New Trends in Databases and Information Systems. AISC, vol. 241, pp. 235–244. Springer, Heidelberg (2014)
51. Przymus, P., Kaczmarski, K., Stencel, K.: A bi-objective optimization framework for heterogeneous CPU/GPU query plans. In: CS&P, pp. 342–354. CEUR-WS (2013)
52. Rabl, T., Poess, M., Jacobsen, H.-A., O'Neil, P., O'Neil, E.: Variations of the star schema benchmark to test the effects of data skew on query performance. In: ICPE, pp. 361–372. ACM (2013)
53. Rauhe, H., Dees, J., Sattler, K.-U., Faerber, F.: Multi-level parallel query execution framework for CPU and GPU. In: Catania, B., Guerrini, G., Pokorný, J. (eds.) ADBIS 2013. LNCS, vol. 8133, pp. 330–343. Springer, Heidelberg (2013)
54. Răducanu, B., Boncz, P., Zukowski, M.: Micro adaptivity in vectorwise. In: SIGMOD, pp. 1231–1242. ACM (2013)
55. Saecker, M., Markl, V.: Big data analytics on modern hardware architectures: a technology survey. In: Aufaure, M.-A., Zimányi, E. (eds.) eBISS 2012. LNBIP, vol. 138, pp. 125–149. Springer, Heidelberg (2013)
56. Sanders, J., Kandrot, E.: CUDA by Example: An Introduction to General-Purpose GPU Programming, 1st edn. Addison-Wesley Professional, Upper Saddle River (2010)
57. Schäler, M., Grebhahn, A., Schröter, R., Schulze, S., Köppen, V., Saake, G.: QuEval: beyond high-dimensional indexing à la carte. PVLDB **6**(14), 1654–1665 (2013)
58. Selinger, P.G., Astrahan, M.M., Chamberlin, D.D., Lorie, R.A., Price, T.G.: Access path selection in a relational database management system. In: SIGMOD, pp. 23–34. ACM (1979)
59. Tsirogiannis, D., Harizopoulos, S., Shah, M.A.: Analyzing the energy efficiency of a database server. In: SIGMOD, pp. 231–242. ACM (2010)

60. Viglas, S.D.: Just-in-time compilation for SQL query processing. PVLDB **6**(11), 1190–1191 (2013)
61. Wu, H., Diamos, G., Cadambi, S., Yalamanchili, S.: Kernel weaver: automatically fusing database primitives for efficient GPU computation. In: MICRO, pp. 107–118. IEEE (2012)
62. Yuan, Y., Lee, R., Zhang, X.: The yin and yang of processing data warehousing queries on GPU devices. PVLDB **6**(10), 817–828 (2013)
63. Zhang, S., He, J., He, B., OmniDB, M.L.: Towards portable and efficient query processing on parallel CPU/GPU architectures. PVLDB **6**(12), 1374–1377 (2013)
64. Zhong, J., He, B.: Medusa: simplified graph processing on gpus. IEEE Trans. Parallel Distrib. Syst. **99**, 1–14 (2013)
65. Zhong, J., He, B.: Parallel graph processing on graphics processors made easy. PVLDB **6**(12), 1270–1273 (2013)