# State-of-the-Art on Query & Transaction Processing Acceleration.

Bernd Amann, Youry Khmelevsky, Gaétan Hains

# State-of-the-Art on Query & Transaction Processing Acceleration

Bernd Amann[1], Youry Khmelevsky[1] and Gaétan Hains[2]

[1]LIP6, Campus Pierre et Marie Curie, Sorbonne Université, Paris
[2]Huawei Technologies, Paris Research Center

**HUAWEI Technical Report CSI-PARIS-TR-2018-10**

2018–09–27

Huawei Technologies
2012Labs/CSI/DPSL/CSI-PARIS



**Huawei Paris R&D Center**

**Abstract**

The vast amount of processing power and memory bandwidth provided by modern Graphics Processing Units (GPUs) make them a platform for data-intensive applications. The database community identified GPUs as effective co-processors for data processing. In the past years, there were many approaches to make use of GPUs at different levels of a database system.

In this Internal Technical Report, based on the [1] and some other research papers, we identify possible research areas at LIP6 for GPU-accelerated database management systems. We describe some key properties, typical challenges of GPU-aware database architectures, and identify major open challenges.

# 1 Introduction [1]

Modern processors are constrained to a certain amount of power they may consume (i.e., the power wall [2]) and further increasing frequency and parallelism would make them overly power hungry. Therefore, hardware 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 [2]. In order to keep up with the performance requirements of the modern information society, tomorrow's database systems will need to exploit and embrace this increased heterogeneity.

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 [3], [4], [5], [6], [7] — and the technology has also found its way into commercial products (e.g., Jedox [8] or ParStream [9]).

Using graphics cards to accelerate data processing is tricky and has several pitfalls: (1) 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. (2) when integrating GPU co-processing into a realworld 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 [10], no unified GPU-aware DBMS architecture has emerged so far.

The authors in [1] found 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.

# 2 Preliminary Considerations

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.

The graphics card itself contains one or more GPUs and a few gigabytes of device memory. 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.

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. 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 a wide variety of devices from multiple vendors [11]. 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 [12].

# 3 Non-Functional and Functional Properties of a GPU-aware DBMS (GDBMS) Architecture

## 3.1 Non-Functional Properties – for which DBMSs are typically optimized for performance and portability

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 [13].

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 [14]. One major point for achieving good performance in a GDBMS is therefore to avoid data transfers where possible.

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). A complex decision model, that incorporates these four factors, is needed to decide on an optimal operator placement [1].

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. In order to achieve optimal performance, each device typically needs its own optimized version of the database operators [15]. We need to take special care to achieve comparable applicability with respect to traditional DBMSs.

## 3.2 Functional Properties

1. 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 [5]. Hence, a GPU-aware database architecture should make heavy use of in-memory technology.
2. A GPU-aware DBMS should use columnar storage. 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 transferred. 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.
3. Tuple-wise processing is not possible on the GPU, due to lacking support for inter-kernel communication. A GDBMS should utilize an operator-at-a-time model.

4. 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, it's possible to expect the hybrid to be faster than a pure CPU- or GPU-resident system.
5. Effective GPU Buffer Management. It is more efficient to transfer few large data sets than many little datasets (with the same total data volume), it could be more beneficial to cache and manage whole columns.
6. 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. 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.
7. 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.

**In Summary:** 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.

# 4 GPU-accelerated DBMS

The following existing DBMS systems use GPU acceleration.

1. CoGaDB (Universitat Magdeburg, 2013, Open Source)
2. GPUDB (Ohio State University, 2013, Open Source)
3. GPUQP (Hong Kong University of Science and Technology, 2007, Open Source)
4. GPUTx (Nanyang Technological University, 2011)
5. MapD (Massachusetts Institute of Technology, 2013)
6. Ocelot (Technische Universitat Berlin, 2013, Open Source)
7. OmniDB (Nanyang Technological University, 2013, Open Source)
8. Virginian, (NEC Laboratories America, 2012, Open Source)

For all eight systems are designed with main-memory databases in mind, keeping a large fraction of the database in the CPU's main memory. GPUQP and MapD also support diskbased data. However, since fetching data from disk is very expensive compared to transferring data over the PCIe bus, 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.

All systems store their data in a columnar layout, there is no system using row-oriented storage. One exception is Virginian.

Most systems support operator-at-a-time bulk processing.

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).

Apart from GPUTx, none of the surveyed GDBMSs support transactions.

The only GDBMSs having a portable, hardware-oblivious database architecture are Ocelot and OmniDB.

## 4.1 Storage Model

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

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

Table 1: Classification of Storage System and Storage Model — Legend:✓— Supported, x — Not Supported, o — Not Applicable

### 4.2 Processing Model

The processing model varies between the surveyed systems. The first observation is that no system uses a traditional tuple-at-a-time volcano model. Most systems support operator-at-a-time bulk processing. The only exception is GPUTx, which does not support OLAP workloads, because it is an optimized OLTP engine.

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

Table 2: Classification of Processing Model — Legend:✓— Supported, x — Not Supported, o — Not Applicable

### 4.3 Query Placement and Optimization

Two major groups of systems: (1) performs nearly all data processing on one processing device (GPUDB, GPUTx, Ocelot, Virginian), whereas group (2) is capable of splitting the workload in parts, which are then processed in parallel on the CPU and the GPU (CoGaDB, GPUQP, MapD, OmniDB). The systems in the first group are 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). The hybrid query optimization approaches of CoGaDB, GPUQP, MapD, and OmniDB are mostly greedy strategies or other simple heuristics. So far, there are no query optimization approaches for machines having multiple GPUs.

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

Table 3: Classification of Query Processing — Legend:✓— Supported, x — Not Supported, o — Not Applicable

### 4.4 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. The trend towards hardware-oblivious DBMSs is likely to continue.

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

Table 4: Classification of Transaction Support and Portability — Legend:✓— Supported, x — Not Supported, o — Not Applicable

**In summary,** most main-memory DBMSs supporting column-oriented data layout and bulk processing to be GPU-accelerated DBMSs. The following extension points can be identified: Cost models, CPU/GPU scheduler, hybrid query optimizer, access structures and algorithms for the GPU, and a data placement strategy. Implementing these extensions is a necessary precondition for a DBMS to support GPU co-processing efficiently.

## 5 Open Challenges and Research Questions

Two major classes of challenges: The IO bottleneck, which includes disk IO as well as data transfers between CPU and GPU, and query optimization.

## 6 Conclusion

The 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 years implementing new co-processing approaches and proposing new

system architectures. 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. The 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.

# References


[1] S. Breß, M. Heimel, N. Siegmund, L. Bellatreche, and G. Saake, *GPU-Accelerated Database Systems: Survey and Open Challenges*, pp. 1–35. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014.

[2] S. Borkar and A. A. Chien, "The future of microprocessors," *Commun. ACM*, vol. 54, pp. 67–77, May 2011.

[3] A. Ailamaki, D. J. DeWitt, M. D. Hill, and M. Skounakis, "Weaving relations for cache performance," in *Proceedings of the 27th International Conference on Very Large Data Bases*, VLDB '01, (San Francisco, CA, USA), pp. 169–180, Morgan Kaufmann Publishers Inc., 2001.

[4] P. Bakkum and K. Skadron, "Accelerating sql database operations on a GPU with CUDA," in *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, GPGPU-3, (New York, NY, USA), pp. 94–103, ACM, 2010.

[5] B. He, M. Lu, K. Yang, R. Fang, N. K. Govindaraju, Q. Luo, and P. V. Sander, "Relational query coprocessing on graphics processors," *ACM Trans. Database Syst.*, vol. 34, pp. 21:1–21:39, Dec. 2009.

[6] J. He, M. Lu, and B. He, "Revisiting co-processing for hash joins on the coupled CPU-GPU architecture," *Proc. VLDB Endow.*, vol. 6, pp. 889–900, Aug. 2013.

[7] H. Pirk, S. Manegold, and M. Kersten, "Waste not... efficient co-processing of relational data," in *2014 IEEE 30th International Conference on Data Engineering*, pp. 508–519, March 2014.

[8] "Palo GPU accelerator," white paper, 2010.

[9] "Parstream — turning data into knowledge," white paper, November 2010.

[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, url = http://doi.acm.org/10.1145/1247480.1247606, year = 2007, bdsk-url-1 = http://doi.acm.org/10.1145/1247480.1247606, bdsk-url-2 = https://doi.org/10.1145/1247480.1247606," in *Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data*, SIGMOD '07, (New York, NY, USA), pp. 1061–1063, ACM.

[11] B. Gaster, L. Howes, D. R. Kaeli, P. Mistry, and D. Schaa, *Heterogeneous computing with openCL: revised openCL 1.* Newnes, 2012.

[12] NVIDIA, "NVIDIA CUDA C programming guide," September 2018.

[13] D. Tsirogiannis, S. Harizopoulos, and M. A. Shah, "Analyzing the energy efficiency of a database server," in *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD '10, (New York, NY, USA), pp. 231–242, ACM, 2010.

[14] B. He, K. Yang, R. Fang, M. Lu, N. Govindaraju, Q. Luo, and P. Sander, "Relational joins on graphics processors," in *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, SIGMOD '08, (New York, NY, USA), pp. 511–524, ACM, 2008.

[15] D. Broneske, S. Breß, M. Heimel, and G. Saake, "Toward hardware-sensitive database operations.," in *EDBT*, pp. 229–234, 2014.