Efficient Cross-Device Query Processing

Holger Pirk[*]
Centrum Wiskunde & Informatica
Science Park 123
Amsterdam, The Netherlands
holger@cwi.nl

ABSTRACT

The increasing diversity of hardware within a single system promises large performance gains but also poses a challenge for data management systems. Strategies for the efficient use of hardware with large performance differences are still lacking. For example, existing research on GPU supported data management largely handles the GPU in isolation from the system's CPU — The GPU is considered the central processor and the CPU used only to mitigate the GPU's weaknesses where necessary. To make efficient use of *all* available devices, we developed a processing strategy that lets unequal devices like GPU and CPU combine their strengths rather than work in isolation. To this end, we decompose relational data into individual bits and place the resulting partitions on the appropriate devices. Operations are processed in phases, each phase executed on one device. This way, we achieve significant performance gains and good load distribution among the available devices in a limited real-life use case. To grow this idea into a generic system, we identify challenges as well as potential hardware configurations and applications that can benefit from this approach.

1. FACING HARDWARE DIVERSITY

Computer Systems don't just become increasingly parallel, they become increasingly diverse as well. Traditional hard disks can be assisted by solid-state disks [4], main memory by hierarchical caches [20] and conventional CPUs by massively parallel extension cards [8]. All these technologies have one thing in common: they perform much better but at a much higher price than their traditional counterparts. A generic strategy to exploit these performance asymmetries for query processing is still lacking. Instead, researchers focus on exploiting the unique properties of each of these devices for a particular subset of operations.

Especially the high compute power of Graphics Processing Units (GPUs) has aroused the interest of data management researchers. Even cheap commodity systems[1] can easily host four GPU extension cards. These extension cards follow a fundamentally different design than traditional CPU-based systems. Fast sequential execution based on behavior prediction (pipelining, prefetching, branch prediction, ...) is replaced by simple, yet massively parallel, execution. The internal memory of these devices is usually orders of magnitude faster but offers much smaller storage capacity than traditional memory. Figure 1 gives a brief overview of the architecture of our test system, hosting two different GPUs.

[*]Advisor: Martin Kersten

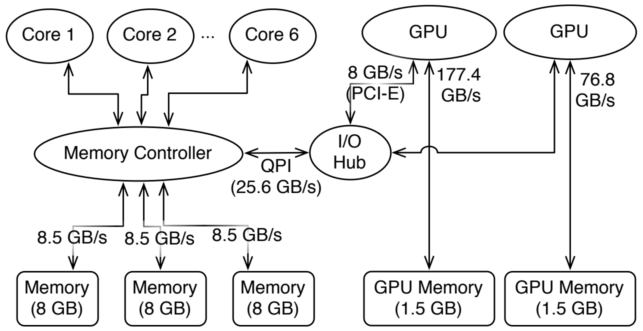


Figure 1: Architecture of a CPU/GPU Computer System

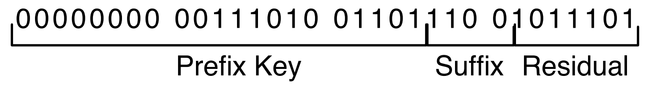
GPU Query Processing

The problem of efficient query processing on the new architecture has received significant attention [3, 9, 14, 7, 15, 12, 8, 13]. However, these efforts usually "re-implemented" a full-fledged relational query processor to run on the massively parallelized architecture. When evaluating queries, relational operators are scheduled onto the most appropriate device. Parallelism (inter- or intra-query) inherent in the workload can be used to balance the load distribution among the devices. However, this approach comes with a number of problems that we discuss in the following.

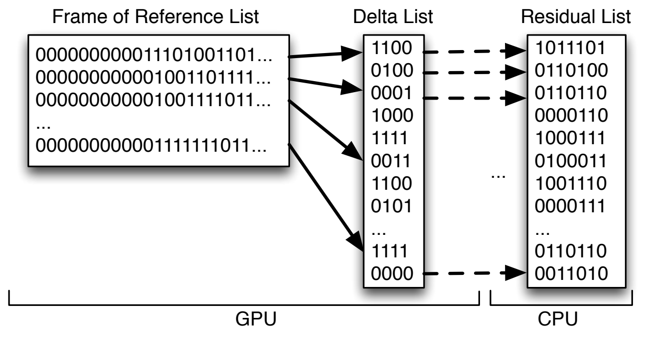
Problems of Existing Approaches

The main problem of the existing approach is that it regards GPUs as processing devices in their own right. When an application does not fit the GPUs capabilities (e.g., when the GPU memory capacity is insufficient), the problems are worked around rather than treated at their core. However, since GPUs are always "hosted" by at least one CPU, they should co-operate rather than work in isolation.

[1]viz. systems with a price of less than $1000

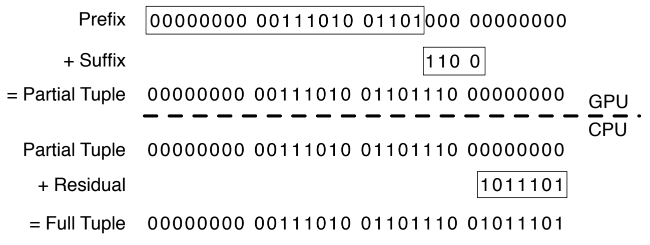


(a) Decomposition



(b) Compression

Figure 2: Bitwise Distribution of Data Among Devices



(a) Two Phase Reconstruction

```
typedef struct {
        short queryID;
        int tupleID;
        char lowResolutionValues[];
} PartialResult;
```

(b) Partial Result

Figure 3: Reconstructing Bitwise Distributed Values

PCI-E Pressure. To process queries on databases that exceed the size of the available device memory, existing systems continuously stream data to and from the respective device. While data may be (partially) cached on the device, generally data is stored in the main memory and transfered to the "appropriate" device through the PCI-E bus. This puts high load on the PCI-E bus, which is generally considered the main bottleneck for CPU/GPU co-processing [11]. The lack of virtual memory paging prevents a "graceful" performance degradation as in CPU-based systems.

Device Underuse. The scheduling granularity of the existing approaches is a relational operator. To make efficient use of multiple devices it is, thus, necessary to have enough independent operators. If the workload does not expose sufficient parallelism, some devices may be underused or even idle. A single user running a group/aggregate query, e.g., will not benefit from having multiple devices available.

Device Overload. Similar in cause, devices may become overloaded if too many operators are regarded as "appropriate" for a single device. E.g., many users scanning a GPU-resident relation will face high latencies due to the lack of independent multithreading on the GPU.

The presented problems indicate that there is still need for improvement in the field of cross device data processing. In the rest of this paper, we present our approach to mitigate the problems (Section 2), present some preliminary results (Section 3) and indicate open problems (Section 4). In Section 5 we, speculate on potential applications and ideas for extensions. We conclude in Section 6.

2. APPROACH

Since the PCI-E bottleneck is the main limiting factor for efficient cross-device processing of large-scale data [11], reducing its load is our primary goal. To avoid the perpetual transfer of data from host to device, we have to reduce the memory footprint of the data on the GPU. To this end, we propose to re-evaluate the techniques that helped reducing the footprint of memory resident databases: the *Decomposed Storage Model (DSM)* [6] reduced the footprint to the columns that are used for query evaluation. Subsequent lightweight compression of the values within a column proved effective [21]. However, to achieve the necessary footprint reduction, decomposing tuples into columns of scalar values may not be enough. To reduce the data volume to the capacity of the GPU memory we propose to take data decomposition to the next level: *Bitwise Decomposition.* Relational tuples/values are (partially) decomposed [18] at the granularity of individual bits. The resulting bit-partitions can be distributed among devices and treated (e.g., compressed) individually. The data footprint on each device can be controlled by varying the number of bits that are stored in the device's memory and tightly packing them across words. Combined with lightweight compression this promises efficient data (pre)processing on devices with limited memory without the need for expensive cross device data transfers.

2.1 Bitwise Distribution

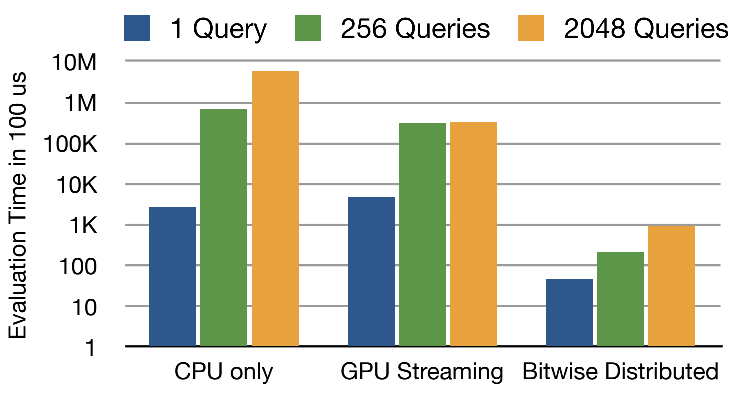
For *natural values*[2], the bits that comprise them have different significance which we exploit to decompose the values and assign the bits to appropriate devices.

Decomposition

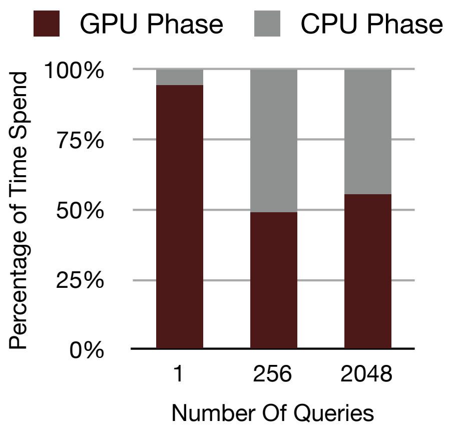
The decomposition and distribution of the individual bits is quite natural (see Figure 2a). Since almost all processing operations have to access the most significant bits, these are stored in the "fastest" memory. The number of bits that are stored on this device should be as high as possible while still leaving space for query evaluation. The data on the GPU is, essentially, a low-resolution representation of the data.

The least significant bits (i.e., residuals) are only necessary to reconstruct the precise values and eliminate false positive results that may result from the processing of the low-resolution data. Since this generally needs less bandwidth than the scanning of the low-resolution data, the least significant bits are stored in the "slower" memory.

[2]Values that directly represent a "measured" value like price or height as opposed to, e.g., categorical values or hashes



(a) Processing Time by Evaluation Technique



(b) Load Distribution for GPU/CPU

Figure 4: Query Evaluation Performance

Compression

Other than reducing the memory footprint, the bitwise decomposition has a second advantage: the low-resolution version is stripped of small-scale variations of data. This makes it highly compressible even with simple, lightweight techniques. To yield even better compression, we physically radix-cluster the low-resolution data and eliminate common prefixes. To achieve low tuple reconstruction costs, we propagate the clustering to the residual values.

Figure 2b provides an illustration of the decomposed, compressed data distributed over a GPU and a CPU. Queries will be processed directly on this representation of the data and exploit the distribution wherever possible.

2.2 Query Processing

The decomposed distribution of the stored tuples largely determines the query processing strategy. Every device is responsible for one phase of the query evaluation. In each phase, one device does the best with the data it has available: narrowing down to the final result as much as possible and (partially) reconstructing the tuple values. While we believe bitwise distribution to be benefitial on any combination of devices with asymmetric capabilities (HDD/GPU, CPU/SSD, ...), we believe that some setups, e.g., HDD/SDD or CPU/GPU, are more "natural" than others. We will focus our work on these "natural" setups.

In a traditional CPU/GPU co-processing setup, queries are evaluated in two phases: *GPU Preselection* and *CPU Refinement*. Figure 3a illustrates the multi-stage reconstruction of a tuple that ends up in the result set. Since the result of each phase is a (potentially inaccurate) representation of tuples in the database, the two phases can be thought of and treated like operators in a relational DBMS. Due to the high overhead when transfering data across devices, the Volcano-model [10] is not well suited to connect these operators. We, thus, connected them using the bulk processing model: in each phase the intermediates are materialized into the device's memory and copied once the phase is finished. When handling continuous query streams, the two evaluation phases of different queries can be interleaved to keep all devices busy.

Phase 1: GPU Preselection

In the first phase, the GPU prefilters the dataset. Since the GPU memory only contains an approximate representation of the data (it misses the residuals), it cannot give an exact answer to the query. Instead it does a best-effort filtering of the tuples and returns partial (low-resolution) results as well as a tuple id to allow reconstruction of the exact value (see Figure 3b). This is equivalent to the early materialization [2] of tuples in a column-oriented database. The partial result set is a superset of the exact answer to the query, but contains all the information for the CPU to narrow it down to the exact result set.

Phase 2: CPU Refinement

In the second phase, the CPU copies the partial results from the GPU's device memory and joins them with the main memory resident residual list. Since this is an invisible/-positional join [1] on the tupleID, it is cheap. The partial results are combined with the residuals to produce the final tuple values (see Figure 3a). The query conditions are evaluated again against the precise values and the results, in case of a hit, copied to the output buffer.

3. PRELIMINARY RESULTS

To evaluate the potential of our approach, we started with an application that is inherently hard to support with secondary indices: Spatial Data Management.

3.1 Spatial Range Queries

The efficient processing of spatial data, is still an open problem. Techniques such as R-Trees, k-d trees or linearization rely on properties of the dataset and queries to perform well. To process arbitrary queries on spatial data, applications often resort to scans of the dataset. Our approach is well suited to improve the performance of such scans. We evaluated Bitwise Distributed Query Processing on a real-life dataset consisting of around 240 Million 2D spatial datapoints that form approximately 450K trajectories (routes taken by users). The data was collected by an industry partner by tracking navigation devices in North-Western Europe. On the database we evaluate a set of rectangular range queries, generated by randomly selecting a point from the dataset and constructing a rectangle around it. The size of the rectangle is random but within a maximum. The generation of the queries is according to a workload description that we received from mentioned industry partner.

We found that the real-life database is hard to compress because the trajectories lack detectable patterns that could be exploited by (lightweight) compression. By decomposing the data, we moved the value variance to the residuals, making the low-resolution representation very compressible.

To support the high degree of processing parallelism in GPUs, we parallelize the query evaluation in two dimensions: the data clusters and the queries. Since, the number of clusters is usually high (tens of thousands), the degree of parallelism supports efficient GPU data processing. The query evaluation follows the paradigm of the bulk execution model that helped to mitigate the per-query overhead for transaction processing on GPUs [16].

Results

The experiments where run on a machine with two Intel® Xeon® CPUs X5650@2.67GHz with 48GB of RAM. The used GPU is a GeForce GTX 480 with 1.5GB of memory.

Figure 4a shows the results of our experiment. CPU is the processing of the queries on the plain main memory resident data. This is the baseline for our evaluation. The state of the art for GPU processing relies on streaming the plain data to the GPU and evaluating the queries in parallel. Whilst the performance compared to CPU-based processing is worse for a single query, the GPU benefits from larger query sets that can be evaluated in parallel. The cross device outperforms all other approaches significantly. For 2048 concurrent queries, GPU/CPU co-processing is more than two orders of magnitude faster than GPU processing on plain data (streaming) and more than three orders of magnitude faster than the CPU only processing.

In addition to the query evaluation performance, bitwise decomposition promises good load balancing over the available devices. To illustrate this, Figure 4b shows the time that is spent processing data on each device. It shows that while single queries induce most of their load on the GPU, the load is almost perfectly distributed for larger query sets.

While these results are encouraging, we believe that there are many more suitable applications as well as unsolved problems.

4. OPEN PROBLEMS

To support more applications, we plan to integrate Bitwise Decomposition into the MonetDB[3] relational Database Management System (DBMS). However, the implementation of a full relational operator set on bitwise decomposed data is not trivial.

4.1 Full Relational Processing

While the performance gain of bitwise distribution for highly selective scans has been studied, the gain for other relational operations is still unclear. To prevent excluding values from the final result, the approximate result has to approach the exact result from the top. The false positives lead to an increase in bandwidth usage which may counteract the achieved performance gains. To mitigate this problem, cross-device equi-joins may benefit from classical distributed join techniques such as gainful semi-joins [5].

Theta-joins on low-resolution data, however, may yield a high number of false positives. Transmitting these to the CPU for accurate processing may consume the time savings that were achieved by using multiple devices. Relational grouping may suffer from the same problem. The impact of this problem is data and application specific and has to be studied. This problem may be approached by increasing the resolution of the low-resolution representation. This does, however, increase the footprint on the GPU which, considering multiple tables, results in an optimization problem.

With an implementation of a full relational query processor at hand, we can evaluate classical analytical workloads as represented by, e.g., the TPC-H and TPC-DS benchmarks. We also plan to evaluate bitwise decomposition in the domain of scientific databases like the SDSS skyserver database [19]. Especially the execution of theta-joins on bitwise distributed data may be benefitial to such applications.

Query Optimization

When evaluating complex queries, the bitwise distribution offers an additional degree of freedom in the query plan. At any operator in the plan, the intermediate result may be either refined or directly used for further processing. While the earlier involves costs for the refinement, the later might multiply the number of false positives. This problem can be tackled using classic query optimization techniques like rule- or cost-based optimization. For that it will be necessary to develop some means to estimate the number of false positives in the approximate result set.

Transaction Processing

To make the approach viable, an efficient strategy for handling updates of the database is required as well. If the updates are only minor corrections of values that only affect the least significant bits, they could be handled by modifying only the residuals. Particularly when writes to the fast memory are expensive, this seems beneficial. Writes to Solid State Disks (SSDs), e.g., slowly degrade the performance and may even hurt the disks reliability. In such cases bitwise decomposition could improve update performance. For coarse grained updates, efficient cross-device update propagation strategies are needed.

4.2 Storage Optimization

Since the size of databases generally exceeds the capacity of the fast device's memory, we are challenged with another problem: which bits should be included in the low-resolution representation and which should be left as residual. If only a single column is stored in the database, the solution is trivial. More attributes form a conflict since the resolution of one attribute can be increased at the expense of another. Picking the optimal resolution for the representation of each attribute is an optimization problem with no obvious solution. We presented an approach to automatically find a cache optimal (partial) decomposition into scalar values for a given database and workload [18]. The used model could be extended to support (partial) bitwise decomposition.

5. APPLICATIONS

We believe that a wide range of applications could benefit from bitwise distribution beyond pure performance.

[3]`http://www.monetdb.org`

Intermediate Visualization

We plan to study how end-user applications may benefit from such multi-stage data processing. An end-user may, e.g., be presented with the approximate result to gain an impression of what he can expect as final result. Based on this, he may even decide that the approximate result is good enough and a refinement unnecessary. The effects are similar to online aggregation [17] but without the inherent reliance on the volcano processing model. This is especially beneficial if the fast processing device is well-connected to an output device. This holds for GPU supported processing as well as, e.g., client-server applications.

Client-Server Applications

In addition to distribution on a single node, we see applications in a client-server environment. A low-resolution representation of the data could be resident on the client or even shipped as part of the application. In a mapping-context, e.g., points of interest could be stored with approximate positions and only located accurately when a user zooms in or moves to a specific area. This would yield a more responsive application with lower bandwidth requirements.

Non-Relational Processing

We believe that the idea of low-resolution preparatory processing and subsequent refinement may be applicable to other domains than relational query processing. Suitable data management challenges can, e.g., be found in multidimensional analytics or sensor data management such as image or video processing. Also the, relatively new, domain of scientific data management may profit from efficient cross device processing. We plan to study such applications.

6. CONCLUSION

Efficient cross-device processing is still an open research challenge. We presented a viable solution, tackling it by decomposing data into individual bits. Our approach outperforms current CPU/GPU co-processing strategies by more than two orders of magnitude for a spatial selection benchmark on real life data. This makes it an attractive subject to study for data processing on multiple devices.

Despite such encouraging results, there is more work to be done. A full relational query processor based on the concept has to be implemented and evaluated. Challenges include transaction processing, query and storage optimization.

In addition, we believe that end-user applications can gain more than mere performance from the approach. Presenting approximate results to an end-user at low latency can significantly improve the user experience.

7. REFERENCES

[1] D. Abadi, S. Madden, and N. Hachem. Column-stores vs. row-stores: How different are they really? In *ACM SIGMOD 2008*, pages 967–980. ACM, 2008.

[2] D. Abadi, D. Myers, D. DeWitt, and S. Madden. Materialization strategies in a column-oriented dbms. In *ICDE 2007*, pages 466–475. IEEE, 2007.

[3] P. Bakkum and K. Skadron. Accelerating SQL database operations on a GPU with CUDA. In *GPGPU-3*, pages 94–103. ACM, 2010.

[4] M. Canim, G. Mihaila, B. Bhattacharjee, K. Ross, and C. Lang. SSD bufferpool extensions for database systems. *VLDB 2010*, 3(1-2):1435–1446, 2010.

[5] M. Chen and P. Yu. Combining joint and semi-join operations for distributed query processing. *IEEE TKDE 1993*, 5(3):534–542, 1993.

[6] G. P. Copeland and S. N. Khoshafian. A decomposition storage model. In *ACM SIGMOD 1985*, pages 268–279. ACM, 1985.

[7] S. Ding, J. He, H. Yan, and T. Suel. Using graphics processors for high performance IR query processing. In *WWW 2009*, pages 421–430. ACM, 2009.

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

[9] W. Fang, B. He, and Q. Luo. Database compression on graphics processors. *VLDB 2010*, 3(1-2):670–680, 2010.

[10] G. Graefe. Volcano-an extensible and parallel query evaluation system. *IEEE TKDE 1994*, 6(1):120–135, 1994.

[11] C. Gregg and K. Hazelwood. Where is the data? why you cannot debate cpu vs. gpu performance without the answer. In *IEEE ISPASS 2011*, pages 134–144. IEEE, 2011.

[12] B. He, W. Fang, Q. Luo, N. Govindaraju, and T. Wang. Mars: a MapReduce framework on graphics processors. In *ACM PACT 2008*, pages 260–269. ACM, 2008.

[13] B. He, N. Govindaraju, Q. Luo, and B. Smith. Efficient gather and scatter operations on graphics processors. In *ACM/IEEE SC 2007*, page 46. ACM, 2007.

[14] B. He, M. Lu, K. Yang, R. Fang, N. Govindaraju, Q. Luo, and P. Sander. Relational query coprocessing on graphics processors. *ACM TODS*, 34(4):21, 2009.

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

[16] B. He and J. X. Yu. High-throughput transaction executions on graphics processors. *Proc. VLDB Endow.*, 4(5):314–325, Feb. 2011.

[17] J. Hellerstein, P. Haas, and H. Wang. Online aggregation. In *ACM SIGMOD Record*, volume 26, pages 171–182. ACM, 1997.

[18] H. Pirk. Cache conscious data layouting for in-memory databases. Master's thesis, Humboldt-Universität zu Berlin, 2010.

[19] A. Szalay, J. Gray, A. Thakar, P. Kunszt, T. Malik, J. Raddick, C. Stoughton, and J. VandenBerg. The sdss skyserver: public access to the sloan digital sky server data. In *ACM SIGMOD 2002*, pages 570–581. ACM, 2002.

[20] M. Zukowski, P. Boncz, N. Nes, and S. Héman. MonetDB/X100-a DBMS in the CPU cache. *IEEE Data Engineering Bulletin*, 1001:17, 2005.

[21] M. Zukowski, S. Heman, N. Nes, and P. Boncz. Super-scalar ram-cpu cache compression. In *ICDE 2006*, pages 59–59. IEEE, 2006.