Column vs. Row Stores for Data Manipulation in Hardware Oblivious CPU/GPU Database Systems

Iya Arefyeva
University of Magdeburg
iia.arefeva@st.ovgu.de

David Broneske
University of Magdeburg
dbronesk@ovgu.de

Marcus Pinnecke
University of Magdeburg
pinnecke@ovgu.de

Mudit Bhatnagar
University of Magdeburg
mudit22.22@gmail.com

Gunter Saake
University of Magdeburg
saake@ovgu.de

ABSTRACT

Finding the right storage model (i.e., row-wise or column-wise storage) is an important task for a database system, because each storage model has its best supported application. Moreover, if we consider the usage of a co-processor (e.g., a GPU), another dimension opens up that influences the selection of the storage model. In fact, factors such as favored memory access pattern of the device and data transfer costs play a vital role in a hybrid CPU/GPU system, influencing the optimal storage model. Since there is currently no evaluation of when to use a column or row store for data manipulation (i.e., we look at insert/update/project operators) in a hybrid CPU/GPU system, we present a framework in OpenCL that we use to investigate the break-even points that determine when to use which storage model.

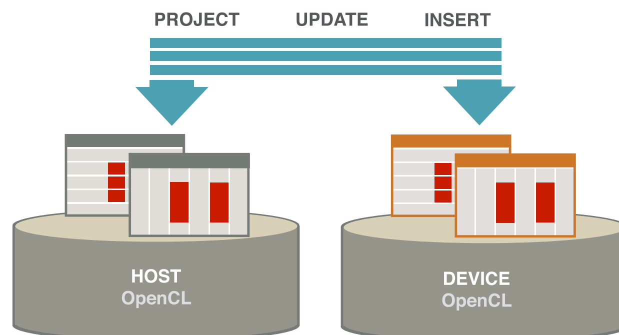

Figure 1: Experimental setup

1. INTRODUCTION

In the literature, there is a big debate about the best storage model for main-memory online transaction processing (OLTP) [5, 13]. The most well-known solution is a delta store [18] that is optimized for insertions relying on a row-wise storage of inserted tuples. In fact, since inserts and deletes work on all attributes of the tuple, a row-wise storage structure is best suited for these operations. In contrast, updates that involve a smaller number of attributes could perform better with a column-wise storage.

Considering the usage of co-processors (e.g., GPUs), several researchers [1, 2, 9, 10] argue for employing a column-wise storage as well, because a column store

- allows for coalesced memory access, which is especially important for GPUs
- has a better compression rate, allowing for more data to be stored in the limited device memory
- can reduce the amount of data to be transferred if only a subset of the columns is needed

However, the main field of application for co-processors is online analytical processing (OLAP)[1]. As a result, it is still unclear what the break-even points between a row-wise and a column-wise storage for co-processor-accelerated OLTP are.

In this paper, we investigate the favored storage model for inserts, updates, and projections on the TPC-C benchmark for a CPU/GPU system implemented in OpenCL (cf. Fig. 1). This builds the basis for further research to state whether column or row stores should be used for co-processor-accelerated OLTP. In particular, we contribute:

- a description of data structures for a column or row store for co-processor acceleration (Section 4.1)
- implementation details of OLTP operators in OpenCL (Section 4.2)
- a first proof-of-concept by evaluating the framework for inserts, updates, and projections (Section 5)

We end this paper with a conclusion and future extensions in Section 6.

2. GPU-ACCELERATED DATA MANAGEMENT

GPU-acceleration has already shown a potential for enabling performance speedups of several cardinalities [1, 9, 14, 15, 16]. Apart from its high potential for compute-intensive tasks [6], GPUs in a co-processing environment are rather limited for I/O intensive applications. Some reasons for that are special architectural challenges that arise when a dedi-

[1]Although GPUTx is an OLTP-centric system using also a column store, there is no evidence whether a row store would hinder transaction processing.

29th GI-Workshop on Foundations of Databases (Grundlagen von Datenbanken), 30.05.2017 - 02.06.2017, Blankenburg/Harz, Germany.
Copyright is held by the author/owner(s).

cated GPU is used in a hybrid system. In fact, there are two aspects that are important: latency and bandwidth between memory of different devices as well as between different memory types of one device. The former is mainly defined by the *transfer overhead* incurred by the PCIe bus, while the latter one is dependent on the used device. For instance, CPUs have a self-managed cache hierarchy that is usually opaque to the user, but GPUs allow for *different memory types* that should be exploited by the programmer to reach peak performance [8]. In the following, we discuss the transfer overhead, memory types and processing model for GPUs.

Transfer overhead.

To process data on the GPU, necessary data has to be shipped from the CPU RAM (Host side) to the GPU RAM (Device side) if it is not already present. Compared to the bandwidth of the on-chip GPU memory (several 100s of GB/s), the bandwidth of the PCIe (currently ca. 16 GB/s; ca. 32 GB/s announced for 2017) is relatively small. Hence, a good data placement is needed to hide this limitation – in fact, data should be distributed on the devices beforehand and operator placement should follow this distribution [3].

GPU memory types.

Apart from the caches that work similar to those of the CPU, the GPU features several other memory types [8]. On a GPU, there are *global, constant, texture, shared* and *local memories* [12, 17]. Overall, the memory types go from a slow but large global memory shared among all compute units of a GPU, over smaller, but cached read-only memory (constant and texture memory) to small but very fast writeable memory shared among processors of a block (shared memory) or compute units of a streaming processor (local memory). Due to the space limitations of each memory type, data has to be loaded carefully into the right memory locations to fully benefit from the GPU acceleration.

GPU processing model.

As mentioned before, a GPU has an internal memory hierarchy that is used for efficient access of data. To make the most out of the available hardware the GPU work group should fetch data from the global memory using a coalesced memory access pattern. Whenever a work group fetches data from the global memory a minimum number of elements are fetched together, hence the work group can utilize this pre-fetched memory block with a fast shared memory. To achieve this optimized execution behavior of the GPU each thread within a work group should access adjacent blocks of memory, this phenomenon where each work item within a work group accesses a sequential block of memory is termed Coalesced Memory Access.

3. RELATED WORK

There are several state of the art databases which use GPU acceleration for either OLAP or OLTP scenarios. Hence in this section, we provide a short overview about their systems and especially the storage schemes used by each.

GPU Accelerated Systems for OLAP.

Considering analytical processing, most of the systems stick to a column-wise storage of data. These systems include GDB by He et al. [9], CoGaDB by Breß et al. [2], Ocelot by Heimel et al. [11]. While GDB relies on processing the queries on the GPU side only, CoGaDB and recent extensions to Ocelot allow the system to process operators either on the GPU or the CPU [4].

GPU Accelerated Systems for OLTP.

To the best of our knowledge, there is only GPUTx as a system working on OLTP processing using GPU acceleration [10]. GPUTx uses a column store, because they argue for a better coalescing of memory access. However, they miss an extensive evaluation in this direction. Hence, our goal is to propose data structures and operator implementations to compare column and row stores for OLTP data manipulation in order to find a suitable storage model for this workload.

4. STORAGE MODEL IMPLEMENTATION

In this section, we present the design choices that were taken to implement a column and row store. To this end, we first introduce the data structures that represent the column and row store. Second, we describe how to implement inserts, updates and projections in a row and column store using OpenCL.

4.1 Data structures

In a row store implementation all values of a tuple are stored next to each other in a contiguous block of memory, followed by the next tuple's values. One implementation of the row store which enables efficient data access is to store the data in an array of type *char*. The length of each attribute's value is fixed and set to the maximum allowed length for this attribute. Therefore, all values of an attribute, regardless of their actual sizes, occupy equal number of bytes. To access an attribute of a tuple, an array of offsets containing the position of each attribute within a tuple has to be passed to the operator. In this way, the operators' implementation is independent of the table schema. Thus, the complete table is represented as a char array of size $N * size_of_a_tuple$, where N is the number of entries.

In a column store, all the values of a column are stored together in one block of memory. To implement this in C++, each column can be represented as a vector containing all the values from the column, thus, each column's values are stored in a contiguous block of memory. Then the complete table can be represented as an instance of a structure that contains all the vectors.

4.2 Operator implementation in OpenCL

OpenCL (Open Computing Language) is an open standard for parallel heterogeneous computing, that can be used with CPUs, GPUs and other devices from different vendors.

We implemented three operators using OpenCL: insert, update and projection. The basic methodology behind the implementation of all three operators is the same with changes in input and output data, as well as the operations performed on the data.

- **Insert**. The input data of the insert operator consists of a table with T entries, where T is the number of tuples to be inserted. For the output table the same amount of memory is allocated as for the input table. The operator copies fields from the input table to the corresponding fields in the output table.

- **Update**. The input data consists of the initial table and a list of positions that should be updated. Attributes, that have numeric type, are increased by 10; text fields get rewritten and replaced by the same data. The operator returns the updated table.

- **Projection**. As input, this operator accepts the initial table and the list of positions of rows, that should be returned, the output data consists of K entries, where K is the number of queries. The operator materializes the attributes of the selected tuples according to their position and writes them to the output data.

In our implementation, the kernels (programs executed on OpenCL devices) for the row store are different from the kernels for the column store, since we use different data structures to represent the tables. For the column store we implemented a separate kernel for each attribute type. In the row store there is only one kernel that is responsible for performing operations on all the attributes.

Row Store Functions.

In Listing 1, we show the functions that we used to access single fields or to store data, from inside the row store kernels. The array **offsets** contains the position of each attribute's value in a tuple, where the first element of the array is always 0 and the last element represents the size of one tuple in bytes. Therefore, the size of this array equals $number_of_attributes + 1$ and can be used to get a tuple's size (lines 2, 9) and to compute the size of an attribute (line 12).

The function **read_value** is used to get a pointer to an element. The pointer to the tuple that contains this element is computed by adding one tuple's size multiplied by the tuple's position (the number of the row) to the pointer to the first element of the whole table (line 3). Then the offset for the required attribute is added to this pointer (line 4).

In the function **write_value** the element's position is computed in exactly the same way. After this step, the new value is written to this position by copying[2] the number of bytes that the value's type takes (line 12).

```
1  global char *read_value(global char *data, int
        tuple_position, int field, global int offsets[], int
        num_of_attributes) {
2      int tuple_size = offsets[num_of_attributes];
3      global char *offset = data + tuple_position *
           tuple_size;
4      offset += offsets[field];
5      return offset;
6  }
7
8  global void write_value(global char *data, int
        tuple_position, char *value, int field, global int
        offsets[], int num_of_attributes) {
9      int tuple_size = offsets[num_of_attributes];
10     global char *offset = data + tuple_position *
           tuple_size;
11     offset += offsets[field];
12     memcpy(offset, value, (offsets[field+1] - offsets[
           field]));
13 }
```

Listing 1: Functions to access or write a value, given its position

[2]The OpenCL language does not provide the function **memcpy**, thus, it has to be implemented manually and added to the kernel.

The whole operator implementation is using the $global_ID$ in order to determine the position of the value that has to be manipulated. Afterwards, the functions **read_value** and **write_value** are used to perform data manipulation at the specified positions according to the three operators.

Column Store Functions.

The column store implementation is straight-forward. For each attribute type, there is a kernel that retrieves its $global_ID$. The $global_ID$ is then used to determine the array position of the data to be manipulated or retrieved.

5. EVALUATION

The operators were evaluated on the CUSTOMER table from the TPC-C benchmark [19] with changes in sizes of some of the text fields. The table's entries consist of 21 attributes, 5 of them are integer numbers, 4 are floating point numbers and 12 attributes are text variables of different length. Both integer and floating point numbers are occupying 4 bytes, the full size of one tuple is 203 bytes. For our experiments we used 30000 entries and the execution time for all the experiments was averaged over 20 runs.

We executed the operators on CPU and GPU using OpenCL for both device executions and we measured the execution time in milliseconds for different number of queries and the following combinations:

- CPU and row store
- CPU and column store
- GPU and row store
- GPU and column store

In our evaluation we used a machine with the following configurations:

- CPU: Intel(R) Core(TM) i5-2500 @3.30 GHz
- GPU: NVIDIA GeForce GT 640
- OpenCL 1.2

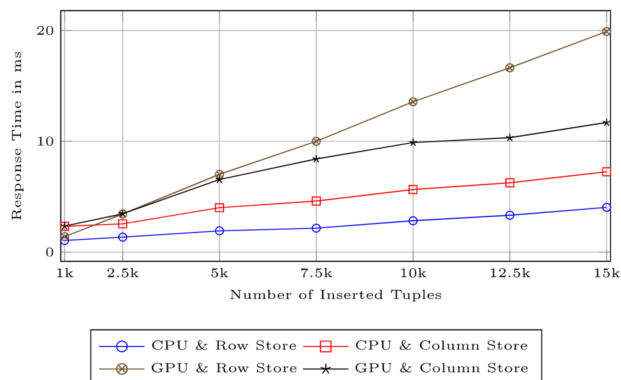

Figure 2: Execution time for the insert operator (incl. transfer time)

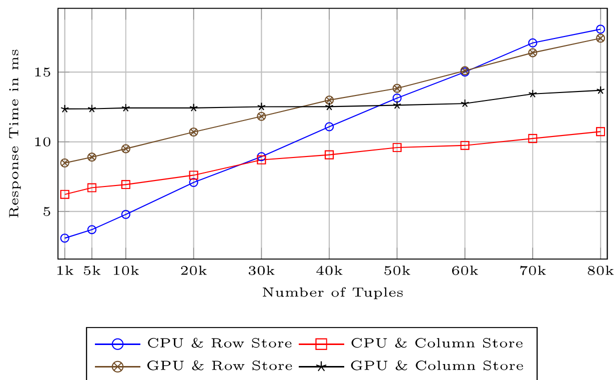

Figure 3: Execution time for the update operator (incl. transfer time)

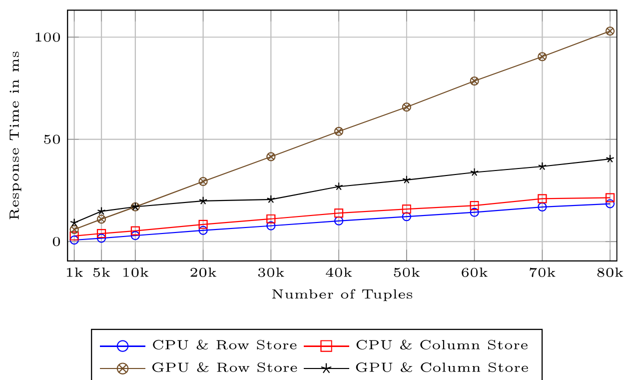

Figure 4: Execution time for the projection operator (incl. transfer time)

Execution time including the transfer time.

In Fig. 2-4, we show the execution time for the insert, update and projection operators respectively including the time for data transfer[3], but excluding the time taken for generating the data and compiling the kernels.

One can note from the figures that for the operators insert and projection (Fig. 2 and 4) the CPU shows the best performance on high numbers of queries independent of the storage model. However, the row store performs better than the column store for inserts and projections on the CPU. In contrast, the more data we insert or project, the more is the row store outperformed by the column store on the GPU. Only for small batch sizes (around few thousands of tuples), a row store storage is beneficial for the GPU. In fact, CPU on a row store is on average 1.5 times faster than the second-best combination (CPU on column store) and almost five times faster than the worst performing combination (GPU on row store).

For the update operator (Fig. 3), column store (on both CPU and GPU) outperforms row store when the number of queries exceeds 25000, however, for 1000 - 25000 queries CPU on row store is faster that the other combinations. CPU and column store handles the data in average 1.5 times faster than GPU and column store. The poor performance of the row store on a big number of update queries is due to the data structure: numeric values are stored in the array of type *char*, so changing these values and writing them back to the array requires two type conversions for each value.

[3]For GPU we measured the time for transferring the data from CPU memory to GPU memory; for CPU it's the time for copying the data inside RAM.

Execution time excluding the transfer time.

Fig. 5-7 shows the time for executing the kernels only.

The general picture stays the same except for the following changes. For the insert operator (Fig. 5), CPU on row store is still 1.5 times faster than CPU on column store, but for the project operator (Fig. 7) row store gets outperformed by column store.

In contrast to the execution time including the transfer time, for the update operator (Fig. 6) GPU on column store performs 1.4 better than CPU.

Overall, the time to transfer the data to the device has an impact on the break-even points that mark when a column-store operator is outperformed by a row-store operator. However, for our evaluated operators, the transfer time is not an exclusive criteria for using either of the storage models.

Execution time for different fractions of the table's columns.

The execution time (including the transfer time) for different fractions of the table's columns was measured for the update and projection operations, launching 50000 and 5000 queries respectively (Fig. 8 and 9), since in real world applications it is rarely needed to update or return the values for all the attributes.

For the update operator, column store has the best performance independent from the number of attributes that are updated. However, for the projection operator the picture is different. When all attributes are selected, CPU and row store performs better than CPU and column store, the same can be observed for GPU. With a decreasing number of attributes, it changes to the opposite: column store shows better performance, because only the columns that need to be returned are transferred. In case of row store, the whole table still needs to be transferred, although only some attributes are processed.

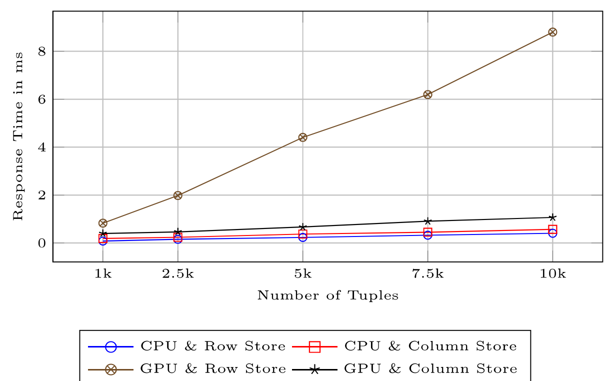

Figure 5: Execution time for the insert operator (excl. transfer time)

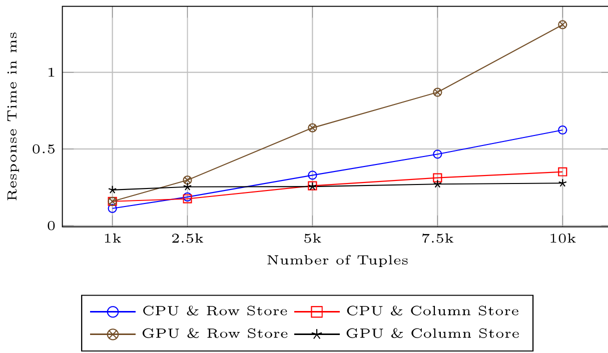

Figure 6: Execution time for the update operator (excl. transfer time)

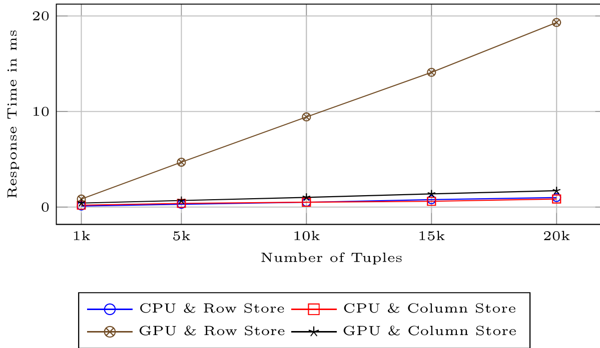

Figure 7: Execution time for the projection operator (excl. transfer time)

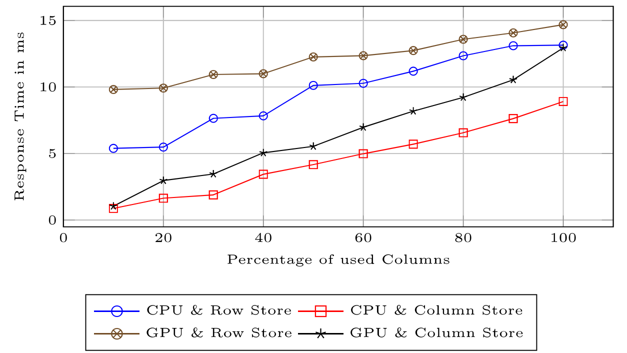

Figure 8: Execution time for the update operator for different fractions of the table's columns

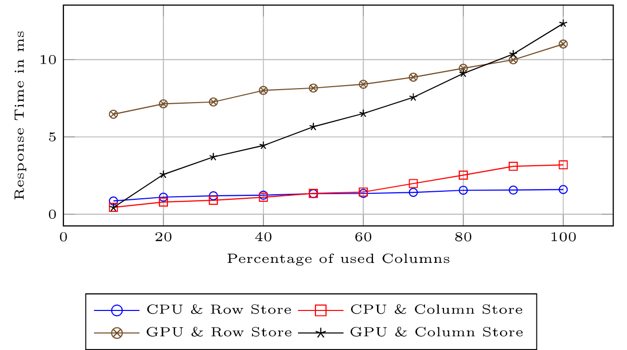

Figure 9: Execution time for the projection operator for different fractions of the table's columns

Evaluation summary.

To summarize our evaluation, we found three interesting facts: (1) Small batch sizes are good for a row store operator on the GPU. (2) For bigger batch sizes, row store operators fall behind the performance of a column store implementation. This is due to a better coalescing when parallelizing a column store operator on the GPU compared to a row store operator, because it has to handle attributes of different sizes. (3) Transfer times only play a vital role for operators that work on a subset of attributes. Hence, the best storage model for insert operators is independent of the transfer time but only depends on the best coalescing for the current implementation.

Still, our current implementation makes some assumptions that may hinder the performance of the row store on the GPU. Especially inserts could be implemented to allow for better coalescing. Currently, inserts work on the granularity of attributes (i.e., float values, integer values and even arrays of chars), which inherently leads to changing offsets for the compute units on neighboring values. As a consequence, the insert operator should be implemented to work on a char granularity. Furthermore, the impact of code optimizations, such as SIMD or loop unrolling, should be further explored [7].

The selection of CPU and GPU for the experiments defines the point, at which one combination is outperformed by a different one. However, the impact of the hardware is expected to become less significant with increasing number of queries.

6. CONCLUSION

Due to the different device properties and application scenarios, the best storage model to be used can vary. In this paper, we investigate the break-even points for inserts, updates and projections in a hybrid CPU/GPU system. Given the data structures and operator implementations in this paper, our results suggest that CPU performs best with a row store and GPU with a column store for inserts and projections. For update operations, a column store seems to be the best storage model for both devices. However, our implementation still leaves some tuning opportunities for the row store open which could boost its performance beyond the one of the column store on the GPU. This opportunity is left open for future work.

7. ACKNOWLEDGEMENTS

This work was partially funded by the DFG (grant no.: SA 465/50-1).

8. REFERENCES

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

[2] S. Breß. The design and implementation of CoGaDB: A column-oriented GPU-accelerated DBMS. *Datenbank-Spektrum*, 14(3):199–209, 2014.

[3] S. Breß, H. Funke, and J. Teubner. Robust query processing in co-processor-accelerated databases. In *SIGMOD*, pages 1891–1906. ACM, 2016.

[4] S. Breß, M. Heimel, M. Saecker, B. Kocher, V. Markl, and G. Saake. Ocelot/HyPE: Optimized data processing on heterogeneous hardware. *PVLDB*, 7(13):1609–1612, 2014.

[5] S. Breß, M. Heimel, N. Siegmund, L. Bellatreche, and G. Saake. GPU-accelerated database systems: Survey and open challenges. *TLDKS*, 15:1–35, 2014.

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

[7] D. Broneske, S. Breß, and G. Saake. Database scan variants on modern CPUs: A performance study. In *In Memory Data Management and Analysis*, pages 97–111. Springer, 2015.

[8] G. Chen, X. Shen, B. Wu, and D. Li. Optimizing data placement on GPU memory: A portable approach. *IEEE TC*, PP(99):1–1, 2016.

[9] 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.*, 34(4):1–39, Dec. 2009.

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

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

[12] A. Meister and G. Saake. Challenges for a GPU-accelerated dynamic programming approach for join-order optimization. In *Proc. GI-Workshop GvDB*, pages 86–81, 2016.

[13] M. Pinnecke, D. Broneske, G. C. Durand, and G. Saake. Are databases fit for hybrid workloads on GPUs? A storage engine's perspective. In *HardBD@ICDE*, 2017.

[14] M. Pinnecke, D. Broneske, and G. Saake. Toward GPU accelerated data stream processing. In *GvDB*, pages 78–83. GI, 2015.

[15] H. Pirk, S. Manegold, and M. Kersten. Accelerating foreign-key joins using asym- metric memory channels. In *VLDB Workshop ADMS*, pages 585–597, 2011.

[16] H. Pirk, S. Manegold, and M. Kersten. Waste not... efficient co-processing of relational data. In *ICDE*, pages 508–519, 2014.

[17] S. Ryoo, C. I. Rodrigues, S. S. Baghsorkhi, S. S. Stone, D. B. Kirk, and W.-m. W. Hwu. Optimization principles and application performance evaluation of a multithreaded GPU using CUDA. In *SIGPLAN PPoPP*, pages 73–82. ACM, 2008.

[18] V. Sikka, F. Färber, W. Lehner, S. K. Cha, T. Peh, and C. Bornhövd. Efficient transaction processing in SAP HANA database: The end of a column store myth. In *SIGMOD*, pages 731–742, 2012.

[19] Transaction Processing Performance Council. tpc-c benchmark revision 5.11. online at
http://www.tpc.org/tpcc/.