Low-Latency Transaction Execution on Graphics Processors: Dream or Reality?

Iya Arefyeva
University of Magdeburg
iya.arefyeva@ovgu.de

Gabriel Campero Durand
University of Magdeburg
campero@ovgu.de

Marcus Pinnecke
University of Magdeburg
pinnecke@ovgu.de

David Broneske
University of Magdeburg
dbronesk@ovgu.de

Gunter Saake
University of Magdeburg
saake@ovgu.de

ABSTRACT

In this paper we take a close look into the role of GPUs for executing OLTP workloads, with a focus on CRUD operator-based processing, as opposed to more complex OLTP transactions. To this end we develop a prototype system supporting GPU and CPU variants of DSM and NSM processing, with a delegation-based approach that uses a single-thread scheduler to manage concurrency control, enabling reads with guaranteed bounded staleness. We evaluate our prototype using workloads from the Yahoo! cloud serving benchmark. We report the impact of layout choices, batching configuration and concurrency control designs. Through our study we are able to pinpoint that the contradicting needs in GPU processing for small batches to reduce waiting time, but large batches to reduce execution time, is the essential challenge for OLTP on these processors, affecting all design choices we study. Hence, we propose two preconditions for supporting OLTP with GPUs, aiming to guide researchers in finding scenarios for extending the applicability of GPUs in supporting data management tasks.

1. INTRODUCTION

In recent years GPUs have transitioned from high-end memory-restricted specialized devices, to omnipresent co-processors, amiable to support general programming even on mobile devices. Such developments have received attention of the database community, leading to the creation of several systems like GPUTx, Ocelot, CoGaDB and Caldera [10, 11, 3, 1].

Save for GPUTx, most systems proposed assume that OLTP, with workloads consisting of high volumes of short transactions, cannot be supported efficiently with GPUs. This assumption is even hard-coded into designs of systems, potentially reducing the role that GPUs can have in the systems. Such choice can be specially a loss for systems supporting hybrid transactional analytical processing (HTAP): if the workload switches to OLTP mostly, GPUs might be underutilized, leading to unsatisfactory distribution of the processing.

Research question: In this short paper we carry out a brief study on the assumption that GPUs are not good matches for OLTP, when compared to CPUs. Specifically we ask: *what are the conditions needed for GPUs to support efficiently OLTP operations?*. We look at this question with a focus on CRUD operator-based processing (in contrast to more complex TPC-C-like transactions).

Contributions: The core contributions of this paper, in seeking to answer our research question can be listed as follows:

- We develop a prototype of a GPU accelerated OLTP system, capable of supporting row-wise and column-wise operations, with concurrency control and support for reads with bounded staleness.
- We evaluate the impact of configurations on pure reads and update-only workloads, showing specifically the large role that batch sizes play in the execution and wait times on GPUs.
- We complement our study with an evaluation of reads with different staleness characteristics. We observe that system-level bounded staleness can increase the throughput on GPUs, to even better extents than when having no concurrency control at all. This observation suggests that studies in supporting requests with bounded staleness for GPU OLTP could be beneficial.
- We conclude by summarizing what we consider to be the essential design challenge for OLTP on GPUs, proposing conditions for addressing it, which could be considered in building GPU-accelerated DBMSs.

This paper is structured as follows: In Sec. 2 we briefly discuss related work, providing the context for our research. In Sec. 3 we present a basic background for our work. In Sec. 4 we describe our implementation. Our evaluation is included in Sec. 5, with a study on the impact of layouts, batch sizes and choice of devices for pure reads and writes workloads. Next we consider the impact of different concurrency control configurations, including strong reads (i.e., a read request guaranteed to see all data committed up until the start of the request), no control and reads with bounded staleness. We conclude our work in Sec. 5.

2. RELATED WORKS

GPU-acceleration for DBMSs: The state of the art, as of 2014, in GPU-accelerated relational systems is surveyed by Breß et al. [4]. He et al. present GPUTx, a relational GPU-accelerated transaction processing system [10]. Stored-procedures aggregated to a single kernel (instead of primitive operators) and the adoption of transactions (via either partition-based or k-set-based lock-free protocols) form the basis of GPUTx. The approach of a k-set transactional protocol (where operations are given the freedom to execute as long as their dependencies are kept) is similar to our design for concurrency control (5.2), however, unlike GPUTx, we prototype a query engine running fine-grained operators, instead of more complex transactions.

Ocelot, as developed by Heimel et al., is a hardware-oblivious version of a GPU-accelerated database [11], stemming from its implementation in OpenCL. Ocelot acts as an extension to MonetDB, offering new operators to the MonetDB query engine. We share with Ocelot the implementation in OpenCL, we diverge, however, since Ocelot does not consider batch-wise concurrency control (i.e., it inherits the optimistic concurrency control from MonetDB) and focuses on OLAP operations. Similarly, CoGaDB [3] is focused on OLAP operations and on the problem of operator placement, aspects that are not specific to our current study.

Mega-KV by Zang et al., is a co-processor accelerated key-value store [14] with the GPU hosting a portion of the data (hashes for keys), and the CPU-memory holding the rest. Authors propose a priority scheduling for collected batches of operators. Priorities match the expected arrival rate, with reads ranking higher than write operations. In our current work we adopt a similar batching of requests, but have not considered prioritization.

More recently, Appaswamy et al. proposed Caldera, a system for heterogeneous transactional and analytical processing using GPU acceleration [1]. As a task distribution approach Caldera uses delegation, with data-to-core assignations, threads-to-transaction mappings, and threads managing concurrency control via explicit message passing. In their system GPUs serve as processors for OLAP workloads, given their massive parallelism; however no consideration is given on the potential of GPUs to serve OLTP operations.

Consequently, we find little to no research on the specifics of operator-based (instead of procedure-based) batched execution for OLTP in GPU-accelerated relational systems, justifying the interest in filling a specific research gap which could help in expanding the role of GPUs in DBMSs.

OLTP benchmarks: There is a wide variety in the design of OLTP benchmarks, from traditional frameworks like TPC-C, relying on complex operations testing write-heavy transactions, to more simple systems like YCSB, which are good for studying specific features of databases (e.g. their concurrency control approach). Difallah et al. present OLTP-Bench [7], a tool for running OLTP benchmarks, encompassing traditional transactional benchmarks (e.g. TPC-C, TATP), web-oriented benchmarks (e.g. epinions, twitter) with graph-like queries, and feature-testing benchmarks (e.g. YCSB, SIBench). For our study we select a benchmark of the latest group (YCSB), since it enables us to focus on the core fine-grained aspects of OLTP on GPUs, such as the impact of batch sizes, concurrency control configuration and layout, on the overall throughput.

Bounded staleness and its evaluation: Systems that manage replicated data are hard-pressed to provide high performance for strong reads (i.e., a read request guaranteed to see all data committed up until the start of the request). To balance the performance penalty for such operations, many systems allow reads with *bounded staleness* (i.e., read operations for which there is a guarantee that the values being read correspond to a version not too behind in time). Research on this topic extends to the late 1980s, with several proposals for formal transactional models to incorporate guarantees for such bounds. Fekete gives a concise overview of the field with a focus on models [8]. Chayka et al. survey alternative measurements proposed in the literature for staleness of individual read requests, or of replicas as a whole [5].

3. BACKGROUND

This section establishes the basic background on row-wise and column-wise storage models, as it pertains to their porting to GPUs. We follow by a concise description of GPU programming and execution models, which serve as a basis to present our evaluation.

3.1 Row vs. Column Store

In *row stores*, tuples are stored in memory contiguously. This allows to quickly perform operations affecting all attributes (e.g., to add/delete a record), making row stores a good fit for OLTP operations. When only a fraction of attributes is needed, row stores might be less efficient since unnecessary fields are mixed together with relevant data [13]. A challenge particular to working with row stores in GPUs is to align memory accesses while supporting variable attribute sizes. However, in recent years hardware alignment is less of a concern for sequential memory accesses as before[1].

In *column stores* all fields of a particular column are stored one after another. All columns of a table may be stored in different memory blocks each, or one after another in a contiguous memory block. This allows to read only the necessary data, providing high performance for operations that affect only values of one or a few columns. Thus, column stores are beneficial for OLAP, which often consists of complex queries that involve aggregations over all values of a column. Additionally, column stores are able to achieve a better compression rate, which allows more data to be stored in memory. This is especially important for GPUs, since their memory size is relatively small compared to their main memory counterpart. The addition of a tuple requires accessing N memory positions individually, when N is the number of attributes in the schema of the table. Since fields of a tuple in row store are located next to each other in memory, they are likely to be pre-fetched in the cache, which, theoretically, makes row stores more efficient for "INSERT" operations.

Given these properties, row stores are well suited for applications that have *record-centric* data access patterns, while column stores are often utilized in systems with an *attribute-centric* data access patterns [13].

3.2 GPU Architecture and Programming Model

Central processing units (CPUs) usually consist of a few powerful cores and are designed for task parallelism, i.e. simultaneous execution of different tasks. CPU processing style is a good fit for OLTP, which involves a big amount of queries, each of them accessing a small number of bytes.

[1]Nvidia researchers point out that in devices of compute capability 2.0, accesses by threads in a warp are coalesced into as few cache lines as possible, reducing, w.r.t previous generations, the impact of misaligned accesses on throughput for sequential access across threads (`https://devblogs.nvidia.com/how-access-global-memory-efficiently-cuda-c-kernels`).

In contrast to CPUs, graphics processing units (GPUs) are composed of multiple cores, and are well suited for data parallelism, i.e. executing tasks, that can be performed on many units of data in parallel. This makes GPUs efficient at executing OLAP queries, since they often require doing the same operation on a lot of fields.

GPUs have several memory types: local, shared, global, constant and texture memories. Each thread has its own local memory, which is visible only to this thread. Threads are grouped into blocks, and can use shared memory, visible only to threads within a block. Global, constant and texture memories are shared across all thread blocks.

A program executed on a GPU is called a *kernel*, it performs operations on one element of the data (e.g., a field or a tuple) and forms a basic unit of parallelism.

The data to process should be sent to the GPU via the PCI-E bus before the execution, and the result is sent back to the main memory after the execution is finished. Another option is to use pinned host memory, which allows to transfer the accessed data from the main memory to the GPU during the execution, since it is guaranteed that the page is not swapped out.

For our implementation, we use OpenCL (Open Computing Language). It is one of the most popular frameworks for heterogeneous programming and is supported by several platforms, including Nvidia, AMD and Intel.

4. DESIGN DECISIONS

This section provides brief description of the storage engine and of the benchmark used for evaluation.

4.1 Framework Design

Our storage engine is implemented in C++, because this language allows to efficiently perform memory manipulations, and also makes it possible to use the standard OpenCL API directly, without the overhead of using third-party APIs.

The data in the engine can be stored either row-wise (in one contiguous array of type *char*) or column-wise (in N such arrays, where N is the number of attributes), and either CPU or GPU can be selected for its processing. One kernel operates one element at a time, for instance, reading 10 tuples requires running the kernel $10*N$ times. In our tests we assume single-sited transactions only.

In case the GPU is used, enough space for the table is allocated in the device memory, and the table is stored there entirely, without storing an additional copy in the RAM. Only the necessary data (e.g., indices and new values) are transferred to the GPU during the processing of workloads. A list of keys and their corresponding rows is maintained by the CPU.

Client requests are handled in a single thread. Whenever a client sends a new request, both the client and the request are saved and stored until the server collects enough requests of a given operator to process a batch. The assignation of request-to-batch is ordered such that there are no conflicts per key (i.e., we adopt a form of conflictless task-scheduling). In case there are no new messages for more than a threshold (in our experiments we decided on a threshold of 100 milliseconds), all the collected requests are processed, because otherwise such cases would deteriorate results.

4.2 Yahoo! cloud serving benchmark

For creating reasonable client requests, we use the Yahoo! cloud serving benchmark (YCSB) [6]. It comes with several predefined workloads consisting of different proportions of *insert*, *read*, *update*, *delete* and short *scan* operations, and allows to implement new workloads. Workloads allow to define the number of records in the table, the proportion and the types of executed operations, and the distribution of requests across the records (Zipfian, uniform or latest). A record in YCSB consists of a key and a set of fields that contain random characters.

Each client sends one request to the server, waits for the reply, and then is able to send the next request. Therefore, several clients are required to process workloads in parallel, for instance, it is necessary to create N clients in order to execute operations in batches of size N.

5. EVALUATION

The following results were obtained by using an Intel Xeon E5-2630 CPU and an Nvidia Tesla K40c GPU. The operating system we use is CentOS 7.1 with kernel version 3.10.0, the OpenCL version is 1.2. All the workloads access a table with 10k entries, where each row consists of 10 attributes of equal size (100 bytes each). The minimum and maximum batch sizes are set to 50 and 500 correspondingly. The maximum throughput in this system, measured by performing no operations on the server, was 71k op/s for a pure read workload and 130k op/s for an update-only workload.

5.1 Pure Reads and Updates

To assess the efficiency of GPUs on short read only and write only operations, we ran two separate workloads. The first workload contains 100k read operations, each requesting all the fields of a row, which results in accessing 1M fields in total. The second workload performs 1M update operations, which change only one field of a row. Both workloads access the entries following a Zipfian distribution.

It can be seen that for workloads consisting only of these operations (Fig. 1(a)-(c)), the combination of CPU and row store provides the best performance, while GPU with column store is consistently the slowest. Small batches prove to be more beneficial than larger ones even for GPUs, although the pure processing time (i.e., time for executing the operations only, excluding batch collection) decreases with increasing batch size (Fig. 1(c) and 2(c)). At small batches, though there is almost no overhead in waiting to collect a batch, the execution on GPUs is inefficient, leading to higher latencies. At larger batches, the execution latency is reduced, at the cost of an increased waiting time. Considering the impact on latency, we observe that fast response per request plays a more important role in determining the total latency than does small processing time.

5.2 Concurrency Control

When operations are executed in batches, they might interfere with one another, and correct results are not guaranteed by default. In order to provide a transactional context, for every new operation we check whether the accessed row has already been accessed by a collected, but not yet executed request. In case the operation interferes with any of the previously received operations, we execute the whole batch that contains the previous one. For instance, if after receiving a new update operation U, we detect that the

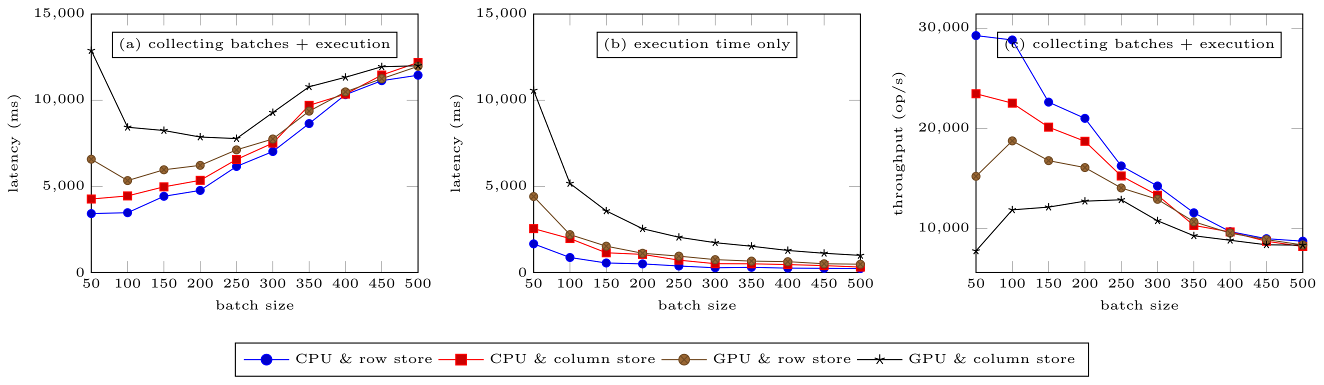

Figure 1: Latency (in ms) and throughput (op/s) for read-only workload.

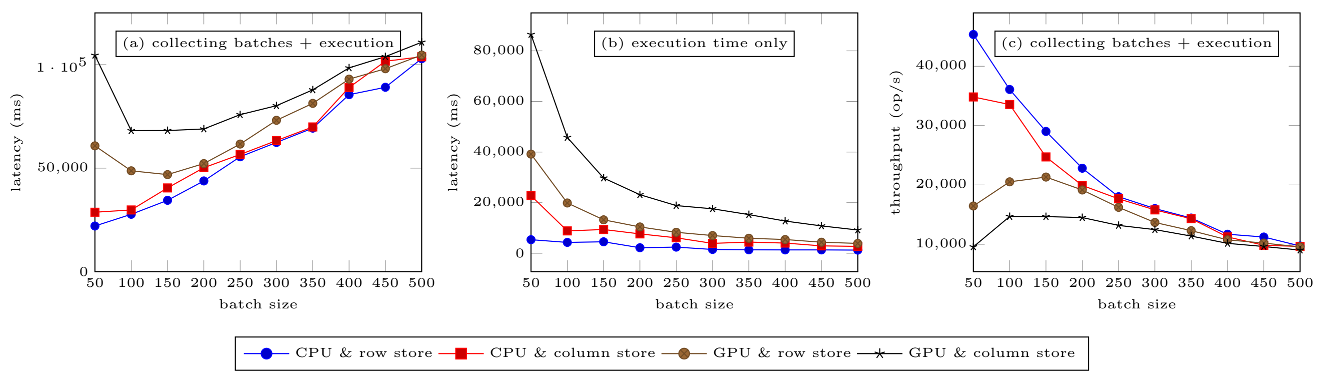

Figure 2: Latency (in ms) and throughput (op/s) for update-only workload.

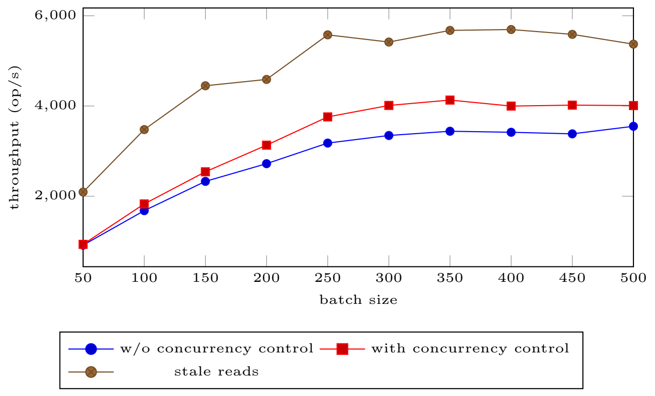

Figure 3: Throughput (op/s) for mixed read and update workload, CPU & row store.

requested row is accessed by a previously collected read operation R, all the read operations are executed. Accordingly we support a basic transactional scheme that does not manage, in the current implementation, transaction failures and rollbacks.

Additionally, in order to analyze, how allowing stale reads would affect the performance, we removed concurrency control for read operations, and let them be executed with a staleness bound of 10 milliseconds (i.e., read operations might not see the writes of operations more recent than 10 milliseconds).

For this evaluation we employed a workload containing 100 k operations, 50% of them being read and the other 50% being update operations. 80% of the operations access entries from the hot set, which consists of the last 20% of entries. Fig. 3-6 shows the throughput for each of the four combinations of devices and storage models in our study.

Enabling concurrency control increased the throughput for the CPU, since it often leads to immediate execution of small batches and hence shorter response times. However, despite the improvements in the waiting time, the GPU's performance is decreased, since the processing of very small batches does not allow to utilize the GPU efficiently, and the increase in the execution time exceeds the time gained by faster responses.

Allowing stale reads is beneficial for all the combinations except for GPU with column store, because read operations are executed without the long waiting time caused by incomplete batches, and hence operations proceed in small batches that increase the already high execution time of the operations on column stores. One might note that for this workload big batches are more beneficial than small ones. Unlike in the pure read and update workloads, the server rarely manages to collect full batches, and thus waits before the execution to make sure that there are no more requests coming. This waiting time is more harmful for small batches, because the server has to wait more often. This issue could

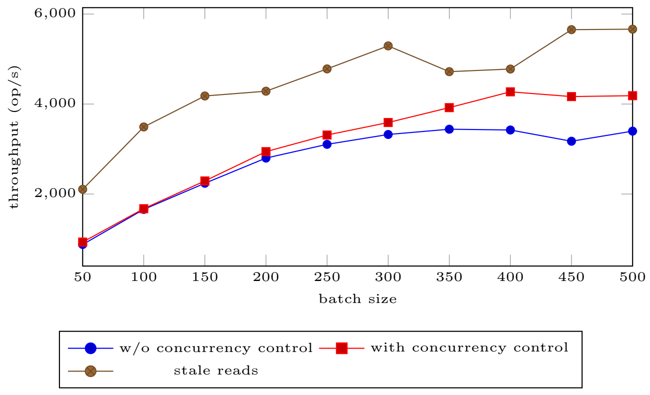

Figure 4: Throughput (op/s) for mixed read and update workload, CPU & column store.

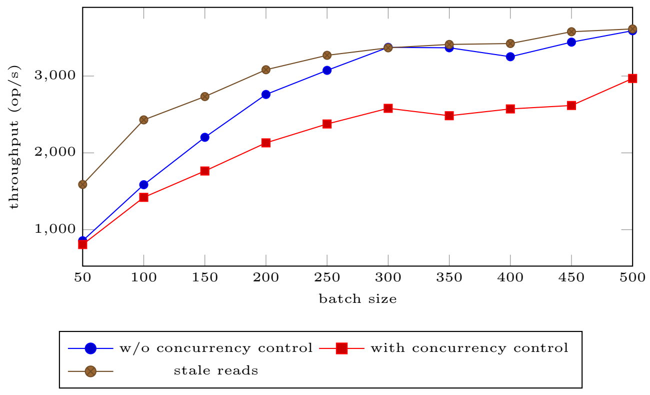

Figure 5: Throughput (op/s) for mixed read and update workload, GPU & row store.

be resolved by adjusting the waiting time spent by the server before executing everything it has collected.

5.3 Discussion

The above described results allow to make the following observations.

CPU and row store outperforms other combinations in both reads and updates, followed by CPU and column store and GPU and row store. GPU with column store seems to provide the worst performance. The difference between the combinations gets less noticeable with increasing batch sizes, because most of the time is taken by handling clients and collecting the batches.

While for the CPU small batches are always more beneficial than big ones, for the GPU the fast response time does not always compensate for the high execution time. One might see that batch size 100 leads to lower latency (Fig. 1(a) and 2(a)) and higher throughput (Fig. 1(c) and 2(c)) than batch size 50, because the GPU is utilized more efficiently, although it takes more time to collect these batches.

Enabling concurrency control (i.e. serving only strong reads) is beneficial for the CPU, since it allows to process smaller batches and reply to clients quicker. Stale reads further improve the performance, because they reduce the waiting time in case of incomplete batches.

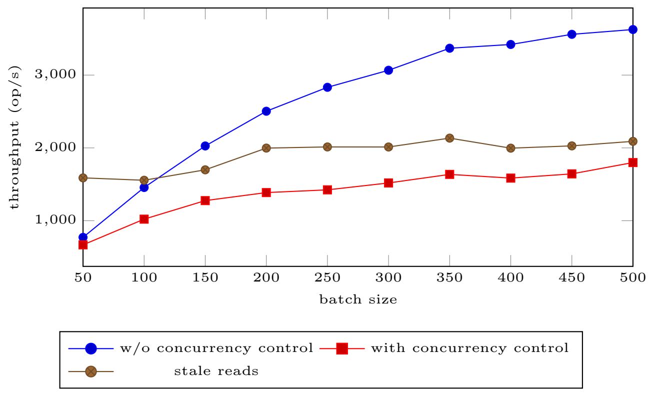

Figure 6: Throughput (op/s) for mixed read and update workload, GPU & column store.

For the GPU processing of smaller batches only decreases the throughput due to the huge loss in the execution speed. However, for GPU with row store allowing stale reads provides better performance than no concurrency control at all.

6. CONCLUSION

In this work, we evaluated the performance of CPUs and GPUs with two storage models (row and column store) using the Yahoo! Cloud Serving Benchmark of OLTP operations.

We can conclude from the experimental results that transaction execution on GPUs is challenging, since one of these two situations always occur:

1. Small batches are processed in order to send results to clients quicker, but processing a small number of elements does not allow to utilize GPUs efficiently.

2. Operations are executed in big batches, which are beneficial for GPUs, but it takes too long to collect these batches and then reply to all the clients.

It is important to note, that in our experiments the entire table was permanently stored on the GPU, thus the performance was evaluated not in the worst case scenario. Transferring the data to the GPU for processing would add an additional overhead, making the usage of the GPU even less efficient. This makes GPUs in their current state not as well-suited for transaction execution as CPUs.

However we should also note that we evaluate a fairly simple transactional context which lacks rollbacks of transactions, and hence realistic overheads are not considered, which could further tilt the balance against GPUs.

In spite of these observations, we argue that OLTP can still be supported with GPUs, provided one of the two following conditions:

1. There is a moderate request arrival rate but it is possible for each request to be broken down to a sufficient amount of parallel operations. One possible case where this occurs could be in comparing vector representations of attributes in tuples (e.g., when fields are represented in a latent vector space, like the case of word embeddings).

2. There is a very high arrival rate of requests, producing little-to-no wait time for forming a large batch of fine-grained operations.

Adding to these conditions, we also observe that the opportunity for reads with bounded staleness is important to boost the efficiency of GPUs. We note that stale reads could either be supported at either system or query level (i.e., when each query defines its own staleness bounds, as proposed for SQL by Guo et al. [9], and as provided for scaling out in systems like Google Spanner [2], and the Asynchronous Parallel Table Replication (ATR) feature of SAP HANA [12]). When co-processor systems adopt such configurations, precise measures for staleness and timeliness (e.g. freshness rate and absolute freshness), in order to enable performance comparisons, should be included in evaluations.

Following our current early observations, in the next stage of our research we will compare our approach for GPU OLTP support with those proposed in the literature and we will further develop our prototype to handle more complex OLTP workloads; as we seek to evaluate potential scenarios and designs that could match the characteristics required for GPUs to work efficiently for OLTP, making GPUs more participative citizens of co-processor accelerated HTAP databases.

7. ACKNOWLEDGMENTS

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

8. REFERENCES

[1] R. Appuswamy, M. Karpathiotakis, D. Porobic, and A. Ailamaki. The case for heterogeneous htap. In *8th Biennial Conference on Innovative Data Systems Research*, number EPFL-CONF-224447, 2017.

[2] D. F. Bacon, N. Bales, N. Bruno, B. F. Cooper, A. Dickinson, A. Fikes, C. Fraser, A. Gubarev, M. Joshi, E. Kogan, et al. Spanner: Becoming a sql system. In *Proceedings of the 2017 ACM International Conference on Management of Data*, pages 331–343. ACM, 2017.

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

[4] S. Breß, M. Heimel, N. Siegmund, L. Bellatreche, and G. Saake. Gpu-accelerated database systems: Survey and open challenges. In *Transactions on Large-Scale Data-and Knowledge-Centered Systems XV*, pages 1–35. Springer, 2014.

[5] O. Chayka, T. Palpanas, and P. Bouquet. Defining and measuring data-driven quality dimension of staleness. Technical report, Università degli Studi di Trento, 2012.

[6] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM symposium on Cloud computing*, pages 143–154. ACM, 2010.

[7] D. E. Difallah, A. Pavlo, C. Curino, and P. Cudre-Mauroux. Oltp-bench: An extensible testbed for benchmarking relational databases. *Proceedings of the VLDB Endowment*, 7(4):277–288, 2013.

[8] A. Fekete. Replica freshness. In *Encyclopedia of Database Systems*, pages 2388–2390. Springer, 2009.

[9] H. Guo, P.-Å. Larson, R. Ramakrishnan, and J. Goldstein. Relaxed currency and consistency: how to say good enough in sql. In *Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, pages 815–826. ACM, 2004.

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

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

[12] J. Lee, S. Moon, K. H. Kim, D. H. Kim, S. K. Cha, and W.-S. Han. Parallel replication across formats in sap hana for scaling out mixed oltp/olap workloads. *Proceedings of the VLDB Endowment*, 10(12):1598–1609, 2017.

[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 *Data Engineering (ICDE), 2017 IEEE 33rd International Conference on*, pages 1599–1606. IEEE, 2017.

[14] K. Zhang, K. Wang, Y. Yuan, L. Guo, R. Lee, and X. Zhang. Mega-kv: a case for gpus to maximize the throughput of in-memory key-value stores. *Proceedings of the VLDB Endowment*, 8(11):1226–1237, 2015.