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

Iya Arefyeva, Gabriel Campero Durand, Marcus Pinnecke, David Broneske, Gunter Saake
Motivation: Context

GPGPUs are becoming essential for accelerating computation

New GPU-Accelerated Supercomputers Change the Balance of Power on the TOP500

- 3 out of Top 5 from HPC 500 (June 2018) are powered by GPUs
- 56% of the flops on the list come from GPU acceleration [10]
Motivation: Context

GPUs are also important for accelerating database workloads:

**Online analytical processing (OLAP):**

- Few long running tasks performed on big chunks of data
- Easy to exploit data parallelism → good for GPUs

*GPU accelerated systems for OLAP:* GDB [1], HyPE [2], CoGaDB [3], Ocelot [4], Caldera [5], MapD [8]

**Online transaction processing (OLTP):**

- Thousands of short lived transactions within a short period of time
- Data should be processed as soon as possible due to user interaction

*GPU accelerated systems for OLTP:* GPUTx [6] → comparably less studied
Motivation: Context

Hybrid transactional and analytical processing (HTAP):

- real-time analytics on data that is ingested and modified in the transactional database engine
- challenging due to conflicting requirements in workloads

GPU accelerated systems for HTAP: Caldera [5]*

*However in Caldera, GPUs don’t process OLTP workloads → possible underutilization
Motivation: GPUs for OLTP

Intrinsic GPU challenges:

1. **SPMD processing**
2. **Coalesced memory access**
3. Branch divergence overheads
4. **Communication bottleneck**: data needs to be transferred from RAM to GPU and back over PCIe bus
5. **Bandwidth bottleneck**: bandwidth of PCIe bus is lower than the bandwidth of a GPU
6. Limited memory

SM structure of Nvidia’s Pascal GP100 SM [9]
Motivation: GPUs for OLTP

OLTP Challenges:
- Managing isolation and consistency with massive parallelism
  - Previous research (GPUTx [6]) proposed a Bulk Execution Model and a K-set transaction handling

Experiments with GPUTx [6]
Our contributions

In this early work, we

1. Evaluate a simplified version of the K-set execution model from GPUTx, assuming single key operations and massive point queries.
2. Test on a CRUD benchmark, reporting impact of batch sizes and bounded staleness.
3. Suggest 2 possible characteristics that could aid in the adoption of GPUs for OLTP, as we seek to adopt them in the design of an GPU OLTP query processor.
Prototype Design
Implementation

- Storage engine is implemented in C++
- OpenCL for GPU programming
- The table is stored on the GPU (in case the GPU is used), only the necessary data is transferred
- Client requests are handled in a single thread
- In order to support operator-based K-sets several cases need to be considered.
  - These cases determine our transaction manager
**Implementation**

**Case 0:** If a batch is not completely filled, the server waits for $K$ seconds after receiving the last request and executes everything ($K=0.1$ in our experiments)

**Case 1: Reads or independent writes:**

<table>
<thead>
<tr>
<th>new request</th>
</tr>
</thead>
<tbody>
<tr>
<td>key 5</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>collected batch for writes</th>
</tr>
</thead>
<tbody>
<tr>
<td>key 10</td>
</tr>
<tr>
<td>key 8</td>
</tr>
<tr>
<td>key 19</td>
</tr>
<tr>
<td>key 4</td>
</tr>
<tr>
<td>key 22</td>
</tr>
<tr>
<td>key 1</td>
</tr>
<tr>
<td>key 56</td>
</tr>
</tbody>
</table>
Implementation

Case 0: If a batch is not completely filled, the server waits for $K$ seconds after receiving the last request and executes everything ($K=0.1$ in our experiments)

Case 1: Reads or independent writes:

```
new request

key 5        write 8
key 10       write 1
key 8        write 2
key 19       write 3
key 4        write 4
key 22       write 5
key 1        write 6
key 56       write 7
key 5        write 8
```

collected batch for writes
Implementation

Case 0: If a batch is not completely filled, the server waits for K seconds after receiving the last request and executes everything (K=0.1 in our experiments)

Case 1: Reads or independent writes:
## Implementation

### Case 2: Write after Write

<table>
<thead>
<tr>
<th>Key</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>write 1</td>
</tr>
<tr>
<td>8</td>
<td>write 2</td>
</tr>
<tr>
<td>19</td>
<td>write 3</td>
</tr>
<tr>
<td>4</td>
<td>write 4</td>
</tr>
<tr>
<td>22</td>
<td>write 5</td>
</tr>
<tr>
<td>1</td>
<td>write 6</td>
</tr>
</tbody>
</table>

New request:

- **Key 4**: write 8

Collected batch for writes!
Implementation

Case 2: Write after Write

new request

| key 10 | write 1 |
| key 8  | write 2 |
| key 19 | write 3 |
| key 4  | write 4 |
| key 22 | write 5 |
| key 1  | write 6 |

collected batch for writes

![batch processing]

| key 10 | write 1 |
| key 8  | write 2 |
| key 19 | write 3 |
| key 4  | write 4 |
| key 22 | write 5 |
| key 1  | write 6 |

flush writes
Case 2: Write after Write

new request

key 4 write 8

key 10 write 1
key 8 write 2
key 19 write 3
key 4 write 4
key 22 write 5
key 1 write 6

flush writes

collected batch for writes

batch processing

key 10 write 1
key 8 write 2
key 19 write 3
key 4 write 4
key 22 write 5
key 1 write 6

collected batch for writes
Case 3: Read after Write

- new request
  - key 4 → read 5

- collected batch for writes
  - key 10 → write 1
  - key 8 → write 2
  - key 19 → write 3
  - key 4 → write 4
  - key 22 → write 5
  - key 1 → write 6

- batch processing
  - key 10 → write 1
  - key 8 → write 2
  - key 19 → write 3
  - key 4 → write 4
  - key 22 → write 5
  - key 1 → write 6

- flush writes

- collected batch for reads
  - key 25 → read 1
  - key 32 → read 2
  - key 13 → read 3
  - key 7 → read 4
  - key 4 → read 5
Implementation

Case 4: Write after Read

new request

key 4  write 5

collected batch for reads

key 10  read 1
key 8   read 2
key 19  read 3
key 4   read 4
key 22  read 5
key 1   read 6
key 56  read 7

flush reads

batch processing

key 10  read 1
key 8   read 2
key 19  read 3
key 4   read 4
key 22  read 5
key 1   read 6
key 56  read 7

collected batch for writes

key 25  write 1
key 32  write 2
key 13  write 3
key 7   write 4
key 4   write 5
Evaluation
YCSB (Yahoo! Cloud Serving Benchmark)

YCSB client architecture [7]
## Workloads

<table>
<thead>
<tr>
<th>Read-only Workload R</th>
<th>Write-only Workload W</th>
<th>Mixed Workload M</th>
</tr>
</thead>
<tbody>
<tr>
<td>100k read operations</td>
<td>1 million update operations</td>
<td>100k read/update operations (50% reads and 50% updates)</td>
</tr>
<tr>
<td>All fields of a tuple are read</td>
<td>Only one field is updated</td>
<td>80% operations access last entries (20% of tuples)</td>
</tr>
<tr>
<td>Zipfian distribution of requests</td>
<td>Zipfian distribution of requests</td>
<td>Goal: What is the impact of concurrency control? Do stale reads improve performance?</td>
</tr>
</tbody>
</table>

**Goal:** Evaluating performance on independent reads or write to find the **impact of batch size**

10k records in the table
Each tuple consists of 10 fields (100 bytes each), key length is 24 bytes

- **CPU:** Intel Xeon E5-2630
- **GPU:** Nvidia Tesla K40c
- OpenCL 1.2
- CentOS 7.1 (kernel version 3.10.0)
Evaluation (workload R - read only)

- CPU & row store provides the best performance
- Small batches reduce collection time
- Very small batches for GPUs are not efficient
- Execution is faster with bigger batches
  - However, it does not compensate for slow response time
Evaluation (workload W - update only)

- CPU & row store provides the best performance
- Small batches reduce collection time
- Very small batches for GPUs are not efficient
- Execution is faster with bigger batches
  - However, it does not compensate for slow response time
Concurrency control is beneficial for the CPU
smaller batches → clients get replies quicker

Allowing stale reads (0.01 s) improves the performance for the CPU due to shorter waiting time before execution

Big batches are better because of the reduced waiting time in case of conflicting operations
big batches→ more operations are executed & the server waits less often
Evaluation (workload M - read/update, GPU)

- Concurrency control is not beneficial for the GPU
  smaller batches → the GPU is not utilized efficiently

- **Allowing stale reads** improves the performance for the GPU & column store due to shorter waiting time before execution

- Big batches are better because of the reduced waiting time in case of conflicting operations
  more operations are executed → the server waits less often
Conclusions and Future Work
The GPU batch size conundrum for OLTP:

**Case 1:** small batches are processed

- clients get replies quicker
- GPUs are not utilized efficiently due to the small number of data elements (this could be improved by splitting requests into fine-grained operations)

**Case 2:** big batches are processed

- many data elements are beneficial for GPUs
- but it takes long to collect batches and throughput can be decreased (this gets faster with higher arrival rates)

+ Other considerations: transfer overhead in case the table is not stored on the GPU
Future Work

- More complex transactions and support for rollbacks
- Concepts for recovery and logging
- Comparison with state of the art systems
Future Work

+ More complex transactions and support for rollbacks
+ Concepts for recovery and logging
+ Comparison with state of the art systems

Thank you!

Questions?
References


8. MapD Product Website: [https://www.mapd.com/](https://www.mapd.com/)


By assuming single operation transactions and no consistency checks, our results avoid divergence and report on larger batches.

We are on single-key CRUD, not SQL level.

All anomalies are preventable through K-sets and managing the dependency graph, so this more general approach supports complete Serializability.

Source: https://blog.acolyer.org/2016/02/24/a-critique-of-ansi-sql-isolation-levels/
Implementation: Introduction

- Two core components:
  - Process Manager (PM): In charge of how processes are executed.
  - Transactional Storage Manager (TxSM): Includes the concurrency control functionality.

From the PM approaches we adopt the latter, but while GP UTx is stored procedure-based, our implementation is operator-based.

Assumptions:
- Break transactions into smaller record-level operators (basic CRUD)
- More complex transactions have to be managed higher in the hierarchy
- composed of many cores
  → multiple threads at a time
- efficient at data parallelism
- limited memory size
- For optimal execution behavior, each thread within a work group should access sequential blocks of memory (coalesced memory access)
- Communication bottleneck: data needs to be transferred from RAM to GPU and back over a PCIe bus
- Bandwidth bottleneck: the bandwidth of a PCIe bus is lower than the bandwidth of a GPU
Motivation: GPUs for OLTP

GPUs for OLTP:

- **The goal:** to reduce the cost of database ownership by improvements in throughput
- There are **challenges** both from the device and from the workload
### Row vs. column store

#### Row store:
- allows to quickly perform operations that affect all attributes
- one pointer to access a tuple
- fields of a tuple are likely to be pre-fetched in the cache
- good fit for OLTP
- if only a fraction of attributes is needed, unnecessary fields are retrieved together with relevant data

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>a1</td>
<td>b1</td>
<td>c1</td>
</tr>
<tr>
<td>a2</td>
<td>b2</td>
<td>c2</td>
</tr>
<tr>
<td>a3</td>
<td>b3</td>
<td>c3</td>
</tr>
</tbody>
</table>

#### Column store:
- allows to read only the necessary data
- good fit for OLAP
- better compression rate
- requires accessing each field separately

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>a1</td>
<td>b1</td>
<td>c1</td>
</tr>
<tr>
<td>a2</td>
<td>b2</td>
<td>c2</td>
</tr>
<tr>
<td>a3</td>
<td>b3</td>
<td>c3</td>
</tr>
</tbody>
</table>