Fundamentals of Transaction Processing on Graphics Cards

Last updated: 17th Jan 2020

M.Sc. Marcus Pinnecke
University of Magdeburg
Institute of Technical and Business Information Systems
http://www.pinnecke.info
Acknowledgements

• These slides are based on the following material:
  – **Co-Processor Accelerated Data Management**
    (Lecture, Sebastian Breß, DFKI)
  – **The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software**
    (Herb Sutter, Dr. Dobb's Journal, 30(3), March 2005)
  – **High-Throughput Transaction Executions on Graphics Processors**
    (Bingsheng He et al., Nanyang Technological University, Singapore)
  – **Professional CUDA C Programming**
    (Book, John Cheng et al., John Wiley & Sons)
  – **The Art of Multiprocessor Programming**
    (Book, Maurice Herlihy et al., Morgan Kaufmann)
  – **Comics inspired by XKCD**

• Valuable feedback and comments came from: Andreas Meister, David Broneske, and Gabriel Campero Durand

Thanks!
M.Sc. Marcus Pinnecke

- **Research associate**
- Institute of Technical and Business Information Systems (ITI)
- Database & Software Engineering Research Group (DBSE)
- Working on a **project** founded by the DFG
- research on **combined transaction processing and analytic processing** involving **heterogeneous compute architectures** (i.e., CPU/GPU/x)
The Free Lunch Is Over
Running Example
Paint walls of a room from white to black

Room (before) → Room (after)
Running Example
Paint walls of a room from white to black

Painter (Thread)
Brush (CPU core)
Color bucket (blocking resource)
Single Threaded Software on Single Core CPU
Improvements in hardware enable higher performance for sequential execution

- **Ever-improving** (compatible) hardware **specification**
  - **CPU Clock Speed**
    - from 300 MHz in 1993 (Pentium) to ~ 4200 MHz in 2016 (Kabylake)
  - **DRAM** (working memory) **Capacity**
    - from 640 KB in 1981 (IBM PC w/ Intel 8088) to 8 TB in 2017 (HPE Integrity Superdome 2 Server)
    - larger **SRAM** (caches and registers) **capacity, disk storage capacity,** ...
- **Optimized clock cycle use:** pipelining, branch predication, out-of-order execution,....
Improvements in hardware enable automatic performance gains in software

- **Effect: Sequential execution gets faster for free**

```c
bool pred(unsigned v); /* returns true iff value v satisfy certain criteria */
unsigned *random_data(unsigned len); /* returns pointer of len random numbers */

unsigned *array = random_data(ARRAY_LEN);
unsigned count = 0;
for (unsigned idx = 0; idx < ARRAY_LEN; idx++)
    count += pred(array[idx]) ? 1 : 0;
```

**Figure:** Determination of the number of elements stored at *array* that satisfy a certain predicate.

- **Optimization for free:** Some optimizations (e.g., loop-unrolling) are turned on by the compiler
The brush increases its size.
The room size remains the same.
The job is done much faster.
Free Performance Lunch (1)

The Free Performance Lunch  Doing **same work** in a **shorter time**

For same workload

- Software improves performance with novel hardware
- Software keeps performance without novel hardware

![Application Performance Graph](image)

- **Sequential, *novel* hardware**
- **Sequential, same hardware**

Same workload
„Software keeps performance without novel hardware“

This means, your old software with your old computer is *today as fast as it was in the past*.

Does this match your experience?
„Software keeps performance without novel hardware“

This means, your old software with your old computer is today as fast as it was in the past.

Your experience is, that your system is getting slower and slower each day, isn’t it?
Room (today)
Room (tomorrow)
Room (day after tomorrow)
The Free Performance Lunch  Doing more work in a fixed time budget

For growing workload

- New software versions keep (or impr.) performance with novel hardware
- New software versions drop performance without novel hardware

**Application Performance**

- Sequential, novel hardware
- Sequential, same hardware
- Sequential, novel hardware
- Sequential, same hardware

Growing workload
Example (Effort render www.google.com; number tags)

- A web browsers **layout engine renders a web page** to display content to users
- Content is (typically) described by (nested) HTML tags
  
  ```html
  <p>Click <a href="...">here</a> for more information</p>
  ```

- Non-visual content (meta-data, structural elements, scripts,...) also by tags
- The **more complex** a web page is, the **more tags** must be processed.
Growing Workload Example (2)

Example (Effort render www.google.com; number tags)

Number of total tags for www.google.com

| Year | # Tags |
|------|--------|
| 1998 | 43     |
| 2000 | 57     |
| 2002 | 73     |
| 2004 | 48     |
| 2006 | 76     |
| 2008 | 95     |
| 2010 | 156    |
| 2012 | 166    |
| 2014 | 161    |
| 2016 | 156    |

Note: Web page contents are loaded from archive.org and archive.org meta-data was removed.
Growing Workload Example (3)

Example (effort to provide service; number of users)

Based on: https://www.statista.com/statistics/264810/number-of-monthly-active-facebook-users-worldwide/

| Year | Number of users (in billions) |
|------|-----------------------------|
| 2009 | 0                           |
| 2010 | 0.5                         |
| 2011 | 1.0                         |
| 2012 | 1.5                         |
| 2013 | 2.0                         |
| 2014 | 2.5                         |
| 2015 | 3.0                         |
| 2016 | 3.5                         |

Number of Active Facebook Users
To keep up with growing workload, is the solution to enjoy free performance lunch by buying new hardware?
Solution?

*(brush increases its size at least as fast as the room increases its size)*
To keep up with growing workload, is the solution to enjoy free performance lunch by buying new hardware?

No.

Consider

(1) the workload might grow faster than performance gains in your software

(2) hard limits in hardware development.
Sequential programs run faster with more recent hardware because

- CPU clock speed increases
- Improved CPU cycle usage
- ....

Today, single CPU clock speed does not significantly grow more

- Development hit practical limits
  for single chip:
  - too much heat
  - too much energy consumption
  - leakage problems

Based on: [http://www.gotw.ca/publications/concurrency-ddj.htm](http://www.gotw.ca/publications/concurrency-ddj.htm)
At some point, the brush size is not practical.
Solution?

(increase number of workers)

(today)
(tomorrow)
(day after tomorrow)
Multi Threaded Software on Single Core CPU
Multi Threaded Software on Single Core CPU

Painters share one brush for painting. They *concurrently* paint on two walls at the same time.

It's faster due to hiding latency w.r.t. moving the color bucket. And there is progress on both walls at the same time.
Free Performance Lunch is Over (2)

Strategy to overcome practical limits for single chip (since 2001):

- **Integrate multiple and independent processing units** („core“) onto **single chip**

![Figure: Uniprocessor](image1.png)

![Figure: Multicore processor](image2.png)
Single Threaded Software on Dual Core CPU
**Important observation:** Sequential software remains sequential; no further speedup*

```c
for (unsigned idx = 0; idx < ARRAY_LEN; idx++)
    count += pred(array[idx]) ? 1 : 0;
```

**Consequences**

- Sequential („single threaded“) software do no longer benefit from CPU development

---

* there is a slight speedup; the operating system’s multitasking feature utilizes multiple cores. Thus, a single process has „more“ from a single core.
Consequence: **The Free Performance Lunch is Over**

- No more performance gains (by clock speed) for free in sequential software

![Application Performance Diagram](image)

**End of Free Performance Lunch**

Without adaption to novel hardware, software do no longer enjoy performance gains*.

* if algorithm complexity does not decrease
Multiple brushes (CPU core) x 2+

Multiple painters (Thread) x 2+
Both painters work concurrently. Each painter has one brush for painting. The act of painting is executed *in parallel* on two walls at the same time.
Software Re-Design in the Multi-Core Age

To keep up with growing workload: Sequential software must be redesigned to match multi-core architecture, i.e., multi-threaded/parallel software is needed.

```c
... parallel_for (array, ARRAY_LEN, [](unsigned idx) {
    count += pred(array[idx]) ? 1 : 0;
});
...```

Core utilization for executing example code

Parallel programming
- Multiple computations simultaneously
- Large problem is split into smaller problems ("tasks")
- Tasks are solved in parallel (e.g., on different cores)
  - Analyzing of data dependency between dependent tasks is important!
Types of Parallelism in Applications

- **Task parallelism (well-suited for CPUs)**, distribution of processing
  - **Different functions** operate on **same or different data**
  - Asynchronous computation
  - Parallelism degree proportional to number of independent tasks
  - Load balancing depends on scheduling algorithm

- **Data parallelism (well-suited for GPUs)**, distribution of data
  - **Same functions** operate on **different subset of same data**
  - Synchronous computation
  - Parallelism degree proportional to input data size
  - Optimal load balancing for multiprocessors
Types of Parallelism

Task Parallelism

Data Parallelism
Types of Parallelism in Applications

Task Parallelisms vs Data Parallelism Example.

Function that takes a memory range starting by `begin` and ending by `end`, and counts the number of elements that satisfy the predicate `pred`.

```c
unsigned count(unsigned* begin, unsigned* end, bool(*pred)(unsigned))
{
    unsigned retval = 0;
    for (unsigned *it = begin; it != end; it++) {
        retval += pred(*it) ? 1 : 0;
    }
    return retval;
}
```

Example (predicate definition):

`pred` could be `bool even(unsigned i) = { return (i % 2 == 0); }`

or also `pred` could be `odd`. 
Types of Parallelism in Applications

Task Parallelisms Example. Determine number of satisfying elements w.r.t. two predicates bool even(unsigned *) and bool odd(unsigned *) on different data unsigned *array1, *array2 by two compute elements (e.g., cores). For each predicate, the number is printed to standard output.

unsigned count_a = 0, count_b = 0;

in parallel /* asynchron */
{
    count_a = count(array1, array1 + ARRAY1_LEN, even);
    count_b = count(array2, array2 + ARRAY2_LEN, odd);
}

printf("#elements satisfy pred_a in array1: %d, #elements satisfy pred_b in array2: %d\n", count_a, count_b);

Note: *array1 and *array2 might point to same data.
**Types of Parallelism in Applications**

**Data Parallelisms Example.** Determine number of satisfying elements w.r.t. one predicate `bool pred(unsigned *)` on **same data** `unsigned *array` by two compute elements (e.g., cores). The number of satisfying elements is printed to standard output.

```c
unsigned count_sum = 0, count_lower = 0, count_upper = 0;

in parallel /* synchron */
{
    count_lower = count(array, array + ARRAY_LEN/2, even);
    count_upper = count(array+ARRAY_LEN, array + ARRAY_LEN, even);
}
```

```c
count = count_lower + count_upper

printf("#elements satisfy pred in array: %d\n", count);
```
Performance Beyond The Free Performance Lunch

To keep up with growing workload: Sequential software must be redesigned to match multi-core architecture, i.e., parallel (multithreaded) software is needed.

Important note: Total clock speed of $n$ cores each having clock speed $c < n \cdot c$
- Multicores introduce communication, coordination, and synchronization overhead
GPGPU Fundamentals
Multiple brushes

Multiple painters

... but each brush favors the same color...

... and painters want to work in groups on the same objects
GPGPU

- GPGPU (General Purpose Computation on Graphics Processing Unit)
  - Generalized GPUs as co-processors beyond graphics processing
    - Compute-intensive tasks (matrix multiplication, query optimization, joining,...)
  - Massive parallelism, and mainly suited for Data Parallelism
  - GPU (device) connected via PCI-Express bus to CPU (host)
  - Popular: Nvidia (CUDA) and AMD (Stream Processors)
    - Dedicated compute model
  - GPU many core architecture
    - GPU: Tesla K40 (Kepler) 2880 cores
    - vs
    - CPU: Intel Core i7 (Sandy Bridge) 4 cores
CUDA GPGPU Program Flow Host/Device

1. Device memory allocation call
2. Memory allocation
3. Copy data to device
4. Invoke function on device
5. Continue host program flow
6. Computation. Maybe invoke further device functions
7. Synchronize with device
8. Copy result back to host
9. Copy (result) data from device to host
10. Cleanup
11. Memory deallocation and device reset

Bottleneck!
Classic CPU vs GPU Purposes & Design

- Control-intensive tasks
- Unpredictable control flow
- Heavyweight thread management
  - 10s of threads on 10s of cores
  - Each thread explicitly managed
- Suites well for Task Parallelism

- Compute-intensive tasks
- Simple control flow
- **Lightweight thread management**
  - 10,000s of threads on 100s of cores
  - Threads managed in batches (blocks)
- Suites well for **Data Parallelism**
CUDA Thread Management

CUDA’s thread hierarchy abstraction

- **Grid**: All threads spawned for single device function
  - Share same **global memory space**
- **Thread Blocks**: Groups of threads in a grid
  - threads **within block**
    - synchronization and shared-memory
  - threads **over block boundaries**
    - no cooperation
- **Single thread** identified by (managed by CUDA)
  - `threadIdx(x, y, z)`: 3D thread index inside block
  - `blockIdx(x, y, z)`: 3D block index inside grid
  - `blockDim(x, y, z)`: 3D Info on block dimension
CUDA Function Invocation

CUDA device function execution declaration and definition

```c
__global__ void device_function_name(/* parameter list */) {
    /* code */
}
```

CUDA C extension: function can be called from host (or device) but is executed on device

CUDA device function execution configuration (from host)

```c
device_function_name <<<numBlocks, numThreadsPerBlock>>> (/* argument list */);
```

definition of grid dimension
definition of block dimension

Example (3,2,5)

|            | Example (7,13,11)                           |
|------------|---------------------------------------------|
| 3          | 7                                           |
| three columns of blocks | seven columns of threads per block |
| 2          | 13                                          |
| two rows of blocks      | thirteen rows of threads per block         |
| 5          | 11                                          |
| layers of blocks        | eleven layers of threads per block         |

Note: For us, usually `numBlocks` and `numThreadsPerBlock` have one dimension only

```c
device_function_name <<<3, 7>>> (/* argument list */);
```
CUDA Data Partitioning

Invocation of device function with (1D) $numBlocks = m$ and (1D) $numThreadsPerBlock = n$

device_function_name $<<<m, n>>>$ (array); /* array of type, in device memory */

Data partitioning

- Array $\ast$array is split into $m$ equal-sized partitions each assigned to one block
- One thread in each block process one of the $n$ values of the block

**Note:** $m$ and $n$ implicitly define the length of $\ast$array to operate on!
By example: The values in a column ("Values") of a relation should be doubled (cf. left).

Host sequential-styled function (definition)

```c
void doubleFn (int *out, int *in, size_t in_len) {
    for (int idx = 0; idx < in_len; idx++) {
        out[idx] = in[idx] * 2;
    }
}
```

Host sequential-styled function (invocation)

doubleFn(result, values, 100);

Host data partitioning and thread assignment
CUDA Function Invocation and Data Partitioning (Device)

By example: The values in a column ("Values") of a relation should be doubled (cf. left).

**Device parallel-styled function (definition)**

```c
__global__ void doubleFn (int *out, int *in) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x
    out[idx] = in[idx] * 2;
}
```

**Device parallel-styled function (invocation)**

```c
doubleFn<5,3>(result, values);
```

**Device data partitioning and thread assignment**

| thread | thread | thread | thread | thread | thread | thread | thread | thread |
|--------|--------|--------|--------|--------|--------|--------|--------|--------|
| 51     | 72     | 16     | 54     | 28     | 90     | 83     | 10     |        |

---

**Fig.: Column „Values”**

**Fig.: Result of `doubleFn` on „Values”**
Transaction Processing on Graphics Processors

(He et al.)
Transaction Processing on GPUs
-or-
How To avoid under-utilization?

Make this blue
Bulk Execution Model

Strategy to Execute Transactions on Graphics Processors

- Proposed in early 2011 by Bingsheng He at International Conference on Very Large Data Bases (VLDB)
- Prototypic implementation as GPUTx using CUDA C
- **Idea:** Bundle a set of transaction (as “bulk”) and execute them as one single kernel on the graphics cards
- **Ingredients:**
  - Pre-defined transaction types as stored procedures
    ```
    Execute procedure_name (arguments)
    ```
  - Transaction ordering by timestamp of submission
  - Transaction Pool, Bulk Profiler and Bulk Generator
  - T-Dependency Graph to model data dependency and guarantee correctness
Bulk Execution Model

Pre-defined transaction types as stored procedures

- Transaction types are conditional executed in single *kernel* function

```c
switch (txn_type) {
    case type_1: /* code */ break;
    case type_2: /* code */ break;
    ...
    case type_n: /* code */ break;
}
```

- A transaction type is a *device* function in CUDA (cannot be called from host)
Bulk Execution Model

Transaction execution

- Submitted **user transactions** are added as **signatures** into **Transaction Pool**:  
  - Unique auto-increment transaction identifier (timestamp)
  - Transaction type name and arguments
- **GPUTx periodically generates a bulk** from **Transaction Pool** for execution  
- **Successful execution**: result are copied from device to host and delivered to user
Correctness of a bulk execution

• Given any initial database, a bulk execution is correct iff the resulting database is the same as that of sequentially executing the transactions in the bulk in the increasing order of their timestamps.
Bulk Execution Model

Definitions

• A transaction \( t \) consists of basic operations \( \text{opp}(X) \) on data item \( X \) in database
  
  – \( \text{read}(X) \) and \( \text{write}(X) \)
  
  – Basic operations have the same timestamp \( \text{time(opp}(X)) \) as \( \text{time}(t) \)

• Conflicting operations. Two basic operations \( \text{opp}_1(X_1) \) and \( \text{opp}_2(X_2) \) conflicts if
  
  – \( X_1 = X_2 \) and \( \text{opp}_1 = \text{write} \) or \( \text{opp}_2 = \text{write} \)

• Conflicting transactions. Two transactions \( t_1 \) and \( t_2 \) conflict if
  
  – \( t_1 \) has an operation that conflicts with an operation in \( t_2 \)

• Conflicting transactions \( t_1, t_2, \ldots, t_n \) must be considered for correct execution
  
  – Determines \( t_1, t_2, \ldots, t_n \) execution order

• Non-conflicting transactions can be executed without concurrency control
Bulk Execution Model

T-Dependency Graph

- Captures data dependency and correctness of bulk execution
- Directed acyclic graph $G$ (i.e., no deadlocks can occur), $T$ set of transactions
  - $i$-th node = transaction $t_i$
  - Edge $(t_x \rightarrow t_y)$ ... data dependency between two transactions $t_x$ and $t_y$
- Edge $(t_x \rightarrow t_y)$ is added iff the following three conditions hold
  (i) $t_x$ and $t_y$ are conflicting
  (ii) $\text{time}(t_x) < \text{time}(t_y)$
  (iii) $\forall t \in T \setminus \{t_x, t_y\} : \text{time}(t_x) \leq \text{time}(t) \leq \text{time}(t_y) \land \text{conflicts}(t, t_x) \land \text{conflicts}(t, t_y)$

**Figure:** Transactions

**Figure:** T-Dependency Graph
**T-Dependency Graph Analysis**

- **Source**: Nodes with no incoming edges are transactions without preceding conflicting transactions

- **Depth (nodes)**: $d(u,s)$ of node $u$ is length of longest from source node $s$ to $u$
  - $d(T_1,T_1) = d(T_2,T_2) = d(T_6,T_6) = d(T_7,T_7) = 0$
  - $d(T_3,T_1) = d(T_5,T_2) = d(T_8,T_7) = 1$
  - $d(T_5,T_1) = 2$

- **Depth (graph)**: $d(G)$ is maximum depth of all nodes in $G$
  - $d(G) = 2$

- **k-Set**: set of nodes with (max) depth $k$
  - $0$-Set = $\{T_1, T_2, T_6, T_7\}$
  - $1$-Set = $\{T_3, T_8\}$
  - $2$-Set = $\{T_5\}$
Bulk Execution Model

T-Dependency Graph Analysis

- **k-Set property (i):** transactions from same k-set are conflict-free.
- **k-Set property (ii):** k-sets must be bulk executed sequentially, i.e., 0-Set; 1-Set; 2-Set;...

**Detail:** k-Set construction is done on graphic card

**Figure:** k-Sets in T-Dependency Graph G
ABSTRACT
OLTP (On-Line Transaction Processing) is an important business sector in various traditional and emerging online services. Due to the increasing number of users, OLTP systems require high throughput for executing tens of thousands of transactions in a short time period. Encouraged by the recent success of GPGPU (General-Purpose computation on Graphics Processors), we propose GPUTx, an OLTP engine performing high-throughput transaction executions on the GPU for in-memory databases. Compared with existing GPGPU studies usually optimizing a single task, transaction executions require handling many small tasks concurrently. Specifically, we propose the bulk execution model to group multiple transactions into a bulk and to execute the bulk on the GPU as a single task. The transactions within the bulk are executed concurrently on the GPU. We study three basic execution strategies (one with locks and the other two lock-free), and optimize them with the GPU features including the hardware support of atomic operations, the massive thread parallelism and the SPMD (Single Program Multiple Data) execution. We evaluate GPUTx on a recent NVIDIA GPU in comparison with its counterpart on a quad-core CPU. Our experimental results show that optimizations on GPUTx significantly improve the throughput, and the optimized GPUTx achieves 4-10 times higher throughput than its CPU-based counterpart on public transaction processing benchmarks.

1. INTRODUCTION
OLTP (On-Line Transaction Processing) is an important business sector generating billions of dollars revenues for database vendors, and even more for online service providers. The market for OLTP is ever growing, as the volume of traditional applications such as credit card services, banking and stock markets becomes larger, and emerging applications including Web 2.0 become popular. In those applications, transaction optimizations are throughput oriented, instead of response time oriented. This calls for a high-throughput transaction execution engine. Most current CPU-based systems [18, 7] adopt in-memory solutions, and utilize multiple CPUs or multiple commodity machines to achieve the performance requirement. Encouraged by the recent success of GPGPU (General-Purpose computation on Graphics Processors), we investigate whether and how we can design a transaction execution engine on the GPU for high throughput.

GPUs (Graphics Processors) have evolved as many-core processors for general purpose computation. They have over an order of magnitude higher memory bandwidth and higher computation power (in terms of GFLOPS) than CPUs. The superb hardware resource enables a GPU to concurrently execute tens of thousands of threads, which can effectively hide the memory latency [8]. This massive thread parallelism is an ideal hardware advantage for transaction executions, where data accesses are usually random and the overall performance is memory latency bounded. While current GPGPU techniques accelerate various database tasks [8, 12], they are mostly limited to a single task such as sort [4] and joins [10], which perform read-only computation on a large volume of data. In comparison, OLTP systems need to handle many small transactions with random reads and updates on the database. Moreover, transaction executions must achieve isolation and consistency for correctness when they perform concurrent updates to the database.

Unique features of transaction processing distinguish itself from the existing GPGPU research. The massive thread parallelism of the GPU poses technical challenges on the correctness and efficiency of transaction executions. On the GPU, executions with thousands of threads are organized in the SPMD (Single Program Multiple Data) execution model. The SPMD execution model has important implications to the efficiency of transaction executions. First,
Wrap Up

Your take-away
Wrap-Up

- **Multi-threaded/parallel software is required**
  - Free Performance Lunch is over
  - Concurrent vs parallel execution
  - Task parallelism and data parallelism

- **Fundamentals of GPGPU programming**
  - Basic graphics cards characteristics
  - CUDA program flow host/device
  - Differences between CPU and GPU
  - Thread hierarchy abstraction
  - Device functions and execution configuration
  - Data manipulation CPU vs GPU

- **Fundamentals of Transaction Processing on GPUs**
  - Bulk execution model
  - Bulk transaction execution
  - Bulk correctness requirements and -checks
  - Bulk data dependencies and T-dependency graph
What happened since then?
Are Databases Fit for Hybrid Workloads on GPUs?
A Storage Engine’s Perspective

Marcus Pinnecke, David Broneske, Gabriel Campero Durand\textsuperscript{1} and Gunter Saake
University of Magdeburg
Email: \{firstname.lastname\}@ovgu.de \& \textsuperscript{1}campero@ovgu.de

Abstract—Employing special-purpose processors (e.g., GPUs) in database systems has been studied throughout the last decade. Research on heterogeneous database systems that use both general- and special-purpose processors has addressed either transaction- or analytic processing, but not the combination of them. Support for hybrid transaction- and analytic processing (HTAP) has been studied exclusively for CPU-only systems. In this paper we ask the question \textit{whether current systems are ready for HTAP workload management with cooperating general- and special-purpose processors}. For this, we take the perspective of the backbone of database systems: the storage engine. We propose a unified terminology and a comprehensive taxonomy to compare state-of-the-art engines from both domains. We show similarities and differences, and determine a necessary set of features for engines supporting HTAP workload on CPUs and GPUs. Answering our research question, our findings yield a resolute: not yet.

I. INTRODUCTION

Two challenges are being set today for database systems: continuous \textit{physical record layout organization} and continuous \textit{compute device assignment} in the face of mixed workload types (cf. Figure 1). On the one hand, database systems need to and SAP HANA [17], address particular challenges implied by the hybridization of both analytical and transactional workload processing into one system. These challenges are: (b.i) different \textit{data access patterns} implied by different workload types,
Memory Management Strategies in CPU/GPU Database Systems: A Survey

Iya Arefyeva, David Broneske, Gabriel Campero, Marcus Pinnecke, and Gunter Saake

University of Magdeburg, Universitätsplatz 2, 39106 Magdeburg, Germany
{ilia.arefeva, dbroneske, campero, pinnecke, saake}@ovgu.de

Abstract. GPU-accelerated in-memory database systems have gained a lot of popularity over the last several years. However, GPUs have limited memory capacity, and the data to process might not fit into the GPU memory entirely and cause a memory overflow. Fortunately, this problem has many possible solutions, like splitting the data and processing each portion separately, or storing the data in the main memory and transferring it to the GPU on demand. This paper provides a survey of four main techniques for managing GPU memory and their applications for query processing in cross-device powered database systems.

Keywords: Cross-device query processing
GPU memory management · Divide-and-conquer · Mapped memory
Unified Virtual Addressing · Unified Memory

1 Introduction

In-memory database systems first were proposed in 1980s [7], and nowadays are becoming more and more popular, as RAM sizes grow and prices are dropping.
Column vs. Row Stores for Data Manipulation in Hardware Oblivious CPU/GPU Database Systems

Iya Arefyeva  
University of Magdeburg  
iiia.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.

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 online analytical processing (OLAP). 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-store stores should be used for co-processor accelerated OLAP.
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 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...
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 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...
What Happened Since Then?

GridTables: A One-Size-Fits-Most H²TAP Data Store
Vision and Concept

Marcus Pinnecke · Gabriel Campero Durand · David Broneske · Roman Zoun · Gunter Saake

Received: date / Accepted: date

Abstract Heterogeneous Hybrid Transactional Analytical Processing (H²TAP) database systems have been developed to match the requirements for low latency analysis of real-time operational data. Due to technical challenges, these systems are hard to architect, non-trivial to engineer, and complex to administrate. Current research has proposed excellent solutions to many of those challenges in isolation - a unified engine enabling to optimize performance by combining these solutions is still missing. In this concept paper, we suggest a highly flexible and adaptive data structure (called GridTable) to physically organize sparse but structured records in the context of H²TAP. For this, we focus on the design of an efficient highly-flexible storage layout that is built from scratch for mixed query workloads. The key challenges we address are: (1) partial storage in different memory locations, and (2) the ability to optimize for mixed OLTP-/OLAP access patterns. To guarantee safe and well-specified data definition or manipulation, as well as fast querying with no redundant storage, we utilize a table scan algorithm and folded row projections.

1 Introduction

In the last decade, the database research community has focused on challenges for data management and system design implied by the ongoing needs to manage and analyze web-scale, frequently changing, diverse datasets. One key challenge is to minimize the latency between operational and analytical systems [34, 48, 45]. For Hybrid Transactional Analytical Processing (HTAP) systems, new architectures were proposed that enable low latency analysis on real-time operational data. A good overview about this topic can be found in a recent survey by Özcan et al. [45]. A key enabling factor for HTAP systems is modern hardware: modern hardware promises novel ways for data processing of relational [15, 25] and non-relational data [42, 50], as well as benefits for several database system components, such as query optimization [26, 41]. Appuswamy et al. even suggested to use the term H²TAP whenever hybridization of workloads is combined with heterogeneity of hardware. [1] For independent database technologies, any attempt to augment or change them should not only preserve existing functionalities but also maintain backward compatibility. This requires an interface that allows to change the structure and behavior of a database system without destroying the data. It is common to implement this by providing a pass-through component, such as a database adapter, which acts as a proxy between the original system and the new one. However, in many cases, the performance requirements of the new interface are significantly different from those of the original one, which makes it difficult to maintain both functionalities simultaneously. Therefore, we propose the concept of GridTable, which allows for efficient storage and retrieval of data in various formats while providing support for advanced analytics functions like parallel processing.
Contribute!
Contribute

Your are **invited to join our research** on modern compute platforms & novel data models:

- **Bachelors or masters thesis**
- **Scientific Project**: Data Management on new Hardware
- **Scientific individual project**

### Some Topic Blocks

- **Modern Engines for Cloud Environments**
- **Self-Tuning**
- **Transaction Execution and Indexing**
- **AI and ML in Database Systems**
- **Document Stores on Modern Hardware**
- **Query Plan Optimization**
- **Benchmarking**
- **Hardware-tuned Operators**