Co-Processor Accelerated Data Management

Sebastian Breß

German Research Center for Artificial Intelligence

Technische Universität Berlin

sebastian.bress@dfki.de

Acknowledgements

These slides are based on the following material:

- Lecture Data Processing on GPUs and GPGPUs (René Müller, IBM Research)
- Lecture Data Processing on Modern Hardware (Jens Teubner, TU Dortmund)
- Tutorial Many-Core-Architekturen zur Datenbankbeschleunigung [14] (Kai-Uwe Sattler and Felix Beier, TU Ilmenau; Jens Teubner, TU Dortmund)

Thanks!

Performance Limitations of Modern Processors

Until 2004/2005:

- Tremendous performance increase of single threaded micro processors
- Two key effects:
  - Number of transistors on a chip doubled every 18 months (Moore's Law [12])
  - Power density of transistors was constant
    $\rightarrow$ Smaller transistors required less voltage and current (Dennard scaling [3])
- DBMSs got faster automatically with faster processors

The Failure of Dennard Scaling

- Transistors got smaller and smaller
- With smaller transistors the leakage current increases
- Higher leakage current results in higher power consumption of transistors

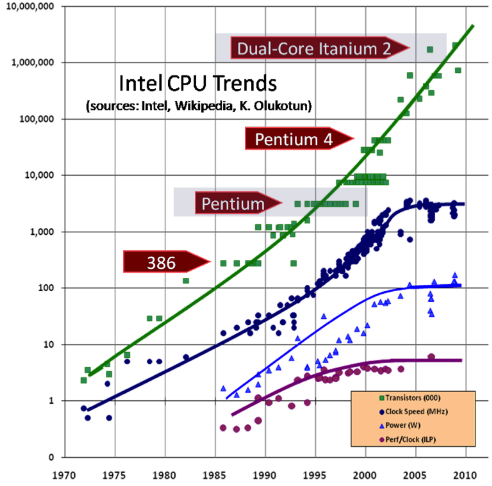
$\rightarrow$ Constant chip size and an increasing number of transistors increases overall power consumption and the produced heat [9]

The Power Wall and the Multi Core Era

- Practical limits on amount of power a processor can use:
  1. Cooling
  2. Energy Consumption
- Consequence: modern processors are limited by a fixed energy budget (power wall) [1]
- In 2004, Intel canceled Tejas and Jayhawk processors
  $\rightarrow$ Since then, they are building multi-core CPUs!

The Free Lunch is Over

Herb Sutter: Dr. Dobb's Journal 30(3), 2005



http://www.gotw.ca/publications/concurrency-ddj.htm

Limitations of Scaling The Number of Cores

- Since 2005, vendors concentrated on processors with multiple cores to use the increasing number of transistors to increase performance
- Benefit of increasing parallelism will not justify the costs of process scaling (i.e., creating even smaller transistors) [4]
- Experts predict that this trend will not scale beyond a few hundred cores [5]
- Examples:
  - Multi-core designs at 22 nm require 21 % of a chip's transistors to be powered off
  - With 8 nm transistors, it will be 50 %! [4]

Dark Silicon and the Heterogeneous Many Core Age

To remain in the power constraints, modern processors can either:

- Operate with lower clock rate or
- Turn off parts of the chip $\rightarrow$ *dark silicon* [1, 4, 5]

How can we exploit the increasing number of transistors?

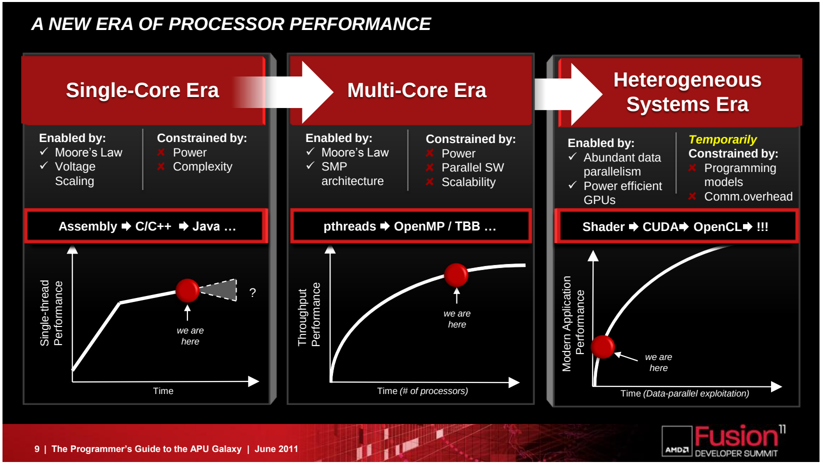
- Provide a large number of cores that are *specialized* for certain tasks
- Cores that are most suitable for the current workload are powered on until the energy budget is reached [4, 5]
- Inevitably, this will cause processors to become increasingly *heterogeneous*

Today’s and Tomorrow’s Processors

- Future machines are expected to consist of a set of heterogeneous processors
- Each processor is optimized for a certain application scenario [1, 5]

This trend has already become commodity in the form of:

- Graphics processing units (GPUs)
- Many integrated cores architectures (MICs)
- Field-programmable gate arrays (FPGAs)
- Accelerated processing units (APUs)
- Processor vendors also combine heterogeneous processors on a single die (e.g., APUs)



A NEW ERA OF PROCESSOR PERFORMANCE

Single-Core Era

Enabled by:
- ✓ Moore's Law
- ✓ Voltage Scaling

Constrained by:
- ✗ Power
- ✗ Complexity

Assembly ➡ C/C++ ➡ Java …

Multi-Core Era

Enabled by:
- ✓ Moore's Law
- ✓ SMP architecture

Constrained by:
- ✗ Power
- ✗ Parallel SW
- ✗ Scalability

pthreads ➡ OpenMP / TBB …

Heterogeneous Systems Era

Enabled by:
- ✓ Abundant data parallelism
- ✓ Power efficient GPUs

***Temporarily* Constrained by:**
- ✗ Programming models
- ✗ Comm.overhead

Shader ➡ CUDA➡ OpenCL➡ !!!

Single-thread Performance
?
we are here
Time
Throughput Performance
we are here
Time (# of processors)
Modern Application Performance
we are here
Time (Data-parallel exploitation)

9 | The Programmer's Guide to the APU Galaxy | June 2011

AMD Fusion11 DEVELOPER SUMMIT

http://developer.amd.com/wordpress/media/2013/06/Phil-Rogers-Keynote-FINAL.pdf

Databases in the Heterogeneous Many Core Age

- Not taking advantage of heterogeneous processors for query processing leaves available resources unused
- On the contrary, we should use these processors to accelerate query processing, e.g., efficient database algorithms:
  - GPUs [6]
  - APUs [7, 8]
  - MICs [10, 11]
  - FPGAs [2, 13]
- We need to think about *how* we can exploit all processors during query processing

Outline

GPGPU for DBMS?

- **tremendous computation power & massive parallelism**

Processor	**FLOPS**	**Cores**
Intel Core i7 (Sandy Bridge)	$\approx$ 102 GFLOPS	4
Tesla K40 (Kepler)	$\approx$ 1.66 TFLOPS	2880

improve performance of

- main-memory DBMS
- compute-intensive tasks: expensive predicates (e.g. for spatial data), UDFs (statistical functions, data mining), query optimization, . . .

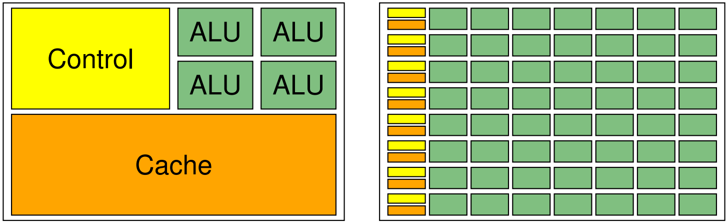
General-Purpose GPUs (GPGPUs)

Original GPU design based on graphics pipeline not flexible enough.

$\rightarrow$ geometry shaders idle for pixel-heavy workloads and vice versa

$\rightarrow$ **unified model** with general-purpose cores

Thus: Design inspired by CPUs, but different

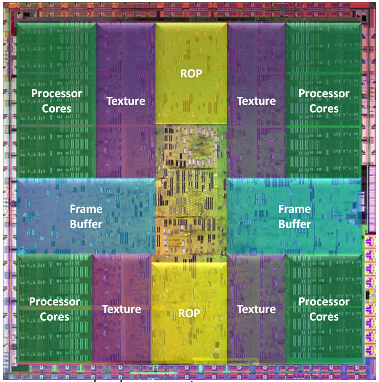


CPU GPU

Rationale: Optimize for **throughput**, not for **latency**.

Example: NVIDIA GPUs

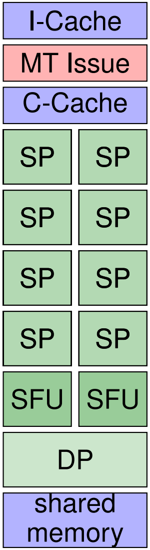
NIVIDA GTX 280



source: www.hardwaresecrets.com

- 10 Thread Processing Clusters
- 10×3 Streaming Multiprocessors
- $10 \times 3 \times 8$ Scalar Processor Cores
  - $\rightarrow$ More like ALUs ($\nearrow$ slide 14)
- Each Multiprocessor:
  - 16k 32-bit registers
  - 16 kB shared memory
  - up to 1024 threads (may be limited by registers and/or memory)

Inside a Streaming Multiprocessor

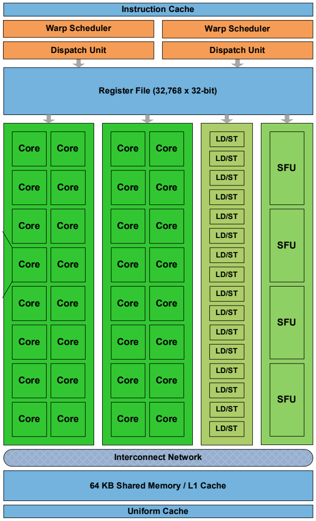


- 8 Scalar Processors (Thread Processors)
  - single-precision floating point
  - 32-bit and 64-bit integer
- 2 Special Function Units
  - sin, cos, log, exp
- Double Precision unit
- 16 kB Shared Memory

- Each Streaming Multiprocessor: up to 1,024 threads.
- GTX 280: 30 Streaming Multiprocessors

  $\rightarrow$ 30,720 concurrent threads (!)

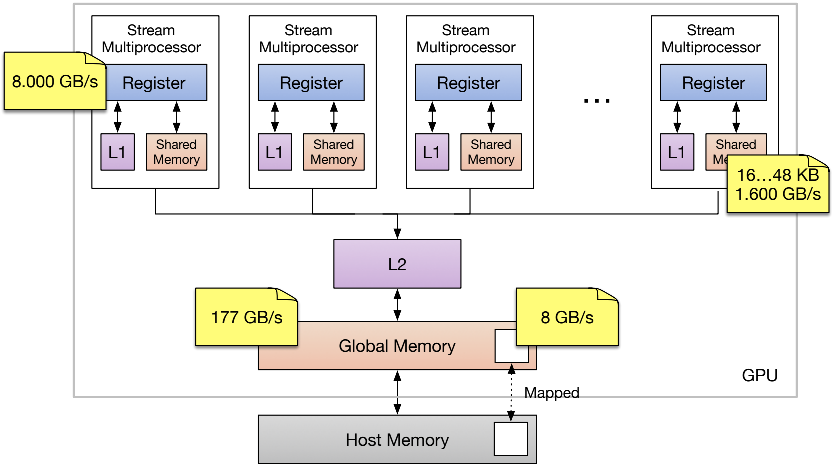
Inside a Streaming Multiprocessor: NVIDIA Fermi



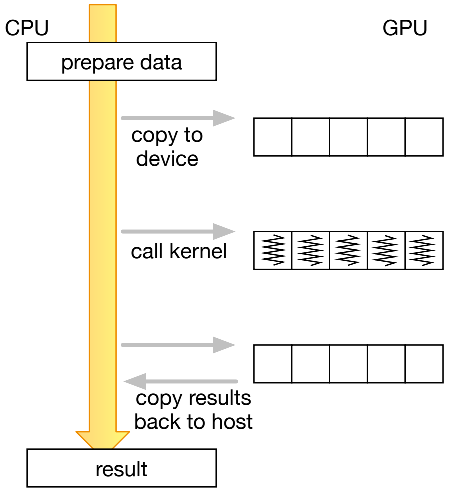
Source: NVIDIA Fermi White Paper

- 32 “cores” (thread processors) per streaming multiprocessor (SM)
- but fewer SMs per GPU: 16 (vs. 30 in GT200 architecture)
- 512 “cores” total
- “cores” now double-precision-capable

GPU: Memory Model



GPU: Execution Model



- data copying host (CPU) $\leftrightarrow$ device (GPU)
- invocations of **compute kernels**
- kernels run asynchronously

Threads on a GPU

To handle 10,000s of threads efficiently, keep things simple.

- Don't try to **reduce** latency, but **hide** it.
  - $\rightarrow$ **Large thread pool** rather than caches
    (This idea is similar to SMT in commodity CPUs)
- Assume **data parallelism** and restrict **synchronization**.
  - $\rightarrow$ Threads and small **groups** of threads use local memories.
  - $\rightarrow$ Synchronization only within those groups (more later).
- Hardware **thread scheduling** (simple, in-order).
  - $\rightarrow$ Schedule threads in **batches** ($\leadsto$ "warps").

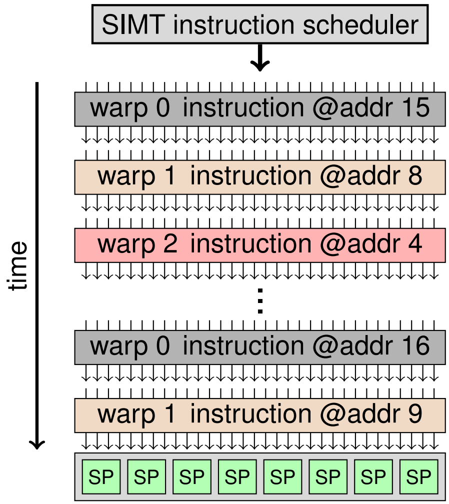
Scheduling in Batches

- In SM threads are scheduled in units of 32, called **warps**.
- **Warp**: Set of 32 parallel threads that start together at the same program address.

warp (dt. Kett- oder Längsfaden)

- Warps are scheduled with zero-overhead
- Scoreboard is used to track which warps are ready to execute
- GTX 280: 32 warps per multiprocessor (1024 threads)
- newer cards: 48 warps per multiprocessor (1536 threads)

SPMD / SIMT Processing


SIMT instruction scheduler
warp 0 instruction @addr 15
warp 1 instruction @addr 8
warp 2 instruction @addr 4
warp 0 instruction @addr 16
warp 1 instruction @addr 9
time
SP SP SP SP SP SP SP SP

- **SIMT**: Single Instruction, Multiple Threads
- All threads execute the same instruction.
- Threads are split into warps by increasing thread IDs (warp 0 contains thread 0).
- At each time step scheduler selects warp ready to execute (*i.e.*, all its data are available)
- NVIDIA Fermi: dual issue; issue two warps at once

Warps and Latency Hiding

Some runtime characteristics:

- Issuing a warp instruction takes **4 cycles** (8 scalar processors).
- Register write-read latency: **24 cycles**.
- Global (off-chip) memory access: $\approx$ **400 cycles**.

Threads are executed **in-order**.

$\rightarrow$ **Hide latencies** by executing other warps when one is paused.

$\rightarrow$ Need **enough warps** to fully hide latency.

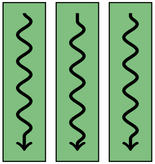
e.g.,

- Need $^{24}/_{4} = 6$ warps to hide register dependency latency.
- Need $^{400}/_{4} = 100$ instructions to hide memory access latency. If every 8th instruction is a memory access, $^{100}/_{8} \approx 13$ warps would be enough.

CPUs vs. GPUs

CPU: task parallelism

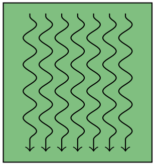
- relatively heavyweight threads
- 10s of threads on 10s of cores
- each thread managed explicitly
- threads run different code



GPU: data parallelism

- lightweight threads
- 10,000s of threads on 100s of cores
- threads scheduled in batches
- all threads run same code

  $\rightarrow$ SPMD, single program, multiple data

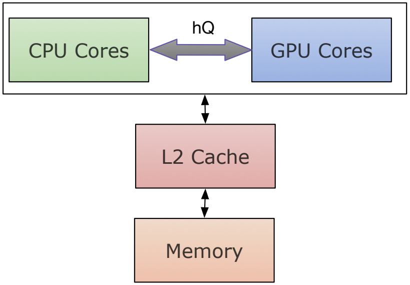


APU: The next step?

CPU + GPU

- GPU kernels are fast
- but GPU has only PCIe bus as I/O path
- PCIe 2.1 / 16 lanes: theoretical throughput of 8 GB/s, but lower in practice $\rightsquigarrow$ **data transfer can overwhelm runtime of kernels**

APU = Accelerated Processing Unit, e.g. AMD A10



- Example: A10-7850K = 12 cores (4 CPU + 6 GPU = 512 execution units), L2 Cache = 4 MB, 32 GB RAM
- CPU and GPU are integrated in the same chip
- . . . and share the L2 cache

Outline

Relational Co-Processing on GPUs

- Selection
- Join
- Aggregation

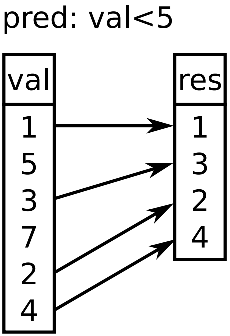
Relational Co-Processing on GPUs

- Selection
- Join
- Aggregation

We focus on selections here.

Selection

Choose a subset of tuples from a relation *R* satisfying a predicate and discard the rest:



```
algorithm:
unsigned int i=0;
for(i=0;i<n;i++){
    if(pred(val[i]))
        res.add(val[i]);
}
```

Parallel Selections

How can we parallelize selections efficiently?

- Concurrent writes may corrupt data structures
  $\rightarrow$ Ensure correctness by latches
- Latching may serialize threads and nullify the performance gained by parallel execution
  $\rightarrow$ Need to ensure correctness without latching
- Key Idea: Pre-compute write locations

B. He et. al. Relational query coprocessing on graphics processors. TODS, 34(4):21, 2009.

Prefix Scans

- Important building block for parallel programs

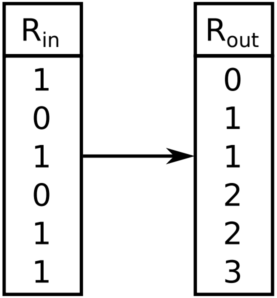
- Applies a binary operator to an array

- Example: prefix sum

- Given an input array R_{in}, R_{out} is computed as follows:
  $R_{out}[i] = R_{in}[0] + \ldots + R_{in}[i-1]$ $(1 \leq i < |R_{in}|)$
  $R_{out}[0] = 0$

B. He et. al. Relational query coprocessing on graphics processors. TODS, 34(4):21, 2009.

Example: Prefix Sum



Parallel Filter

- Create an array *flags* of the same size as *R* and init with zeros
- For each tuple set corresponding flag in *flags* if and only if the current tuple matches the predicate
  $\rightarrow$ *flags* array contains a 1 if the corresponding tuple in *R* is part of the result
  $\rightarrow$ The sum of the values in *flags* is the number of result tuples $\#rt$
- Compute the prefix sum of *flags* and store it in array *ps*
  $\rightarrow$ Now we have the write locations for each tuple in the result buffer

B. He et. al. Relational query coprocessing on graphics processors. TODS, 34(4):21, 2009.

Parallel Filter

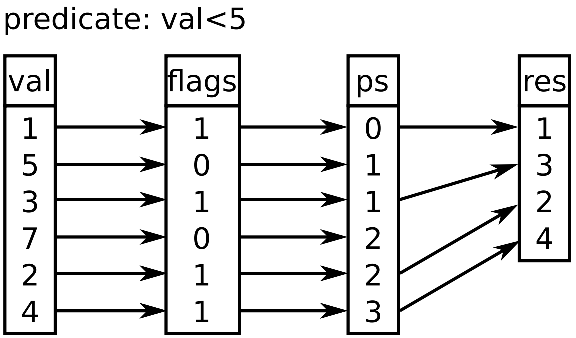
- Create the result buffer *res* of size $\#rt$

- Scan *flags*: if(flags[i]==1) write R[i] to position ps[i] in the result buffer:

```
do in parallel:
 for(unsigned int i=0;i<n;i++){
   if(flags[i]==1){
     unsigned int res_write_index=ps[i];
     res[res_write_index]=R[i];
   }
 }
```

B. He et. al. Relational query coprocessing on graphics processors. TODS, 34(4):21, 2009.

Parallel Filter: Example



```
build flag array:
if(pred(val[i]))
   flags[i]=1;
else
   flags[i]=0;
```

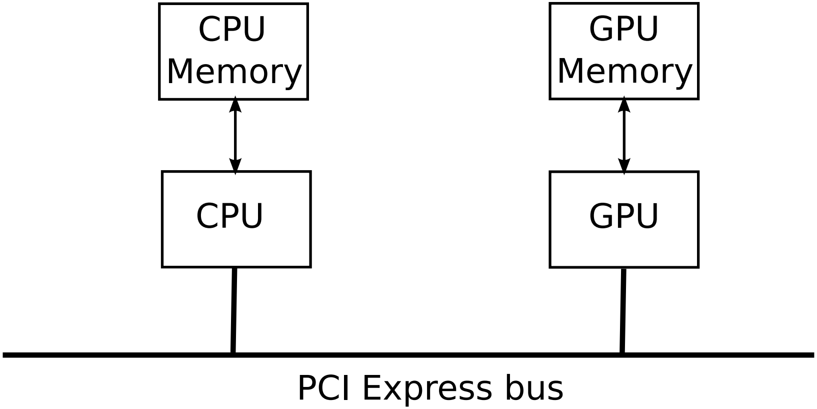
compute prefix sum *ps* from *flags*

scan *flags* and write val[i] to position ps[i] in result array

B. He et. al. Relational query coprocessing on graphics processors. TODS, 34(4):21, 2009.

Outline

Communication Bottleneck



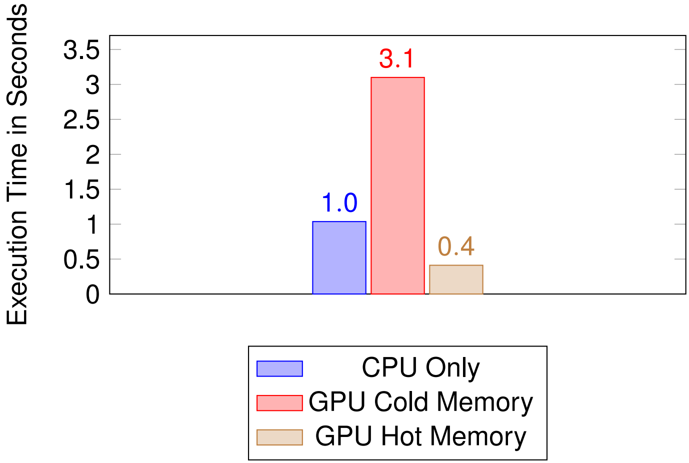
Resource Limitations

- Co-Processor's memory capacity quite small (up to 16 GB)
  $\rightarrow$ Cannot fit all data on the co-processor

- Ad-hoc data transfers are expensive
  $\rightarrow$ Cache input data in co-processor memory and process data locally as much as possible

S. Breß. H. Funke, J. Teubner. Robust Query Processing in Co-Processor-accelerated Databases. In SIGMOD, 2016.

Cache input data for best performance



S. Breß. H. Funke, J. Teubner. Robust Query Processing in Co-Processor-accelerated Databases. In SIGMOD, 2016.

Cache Thrashing

Background:

- Assume a fixed set of re-occurring queries (for discussion only)
- When a query accesses data, it is cached on the co-processor
- If cache is full, data from other queries is evicted

Effects:

- Occurs when the working set does not fit into data cache of a co-processor
- Queries evict each others data from the cache
- In the worst case, we never have a cache hit (if every query accesses different data)
  $\rightarrow$ We pay the full data transfer cost for every query despite having a cache!

S. Breß. H. Funke, J. Teubner. Robust Query Processing in Co-Processor-accelerated Databases. In SIGMOD, 2016.

Operator Placement and Data Placement

Operator Placement:

- Select a processor for each operator in the query plan
- Aims to increase data locality in a single query

Data Placement:

- Select which part of the database is stored in a co-processor's memory
  $\rightarrow$ Reserve part of the co-processors device memory as data cache
- Aims to increase data locality across queries
- Similar to classical page replacement strategies in disk-based databases

S. Breß. H. Funke, J. Teubner. Robust Query Processing in Co-Processor-accelerated Databases. In SIGMOD, 2016.

Duality of Operator and Data Placement

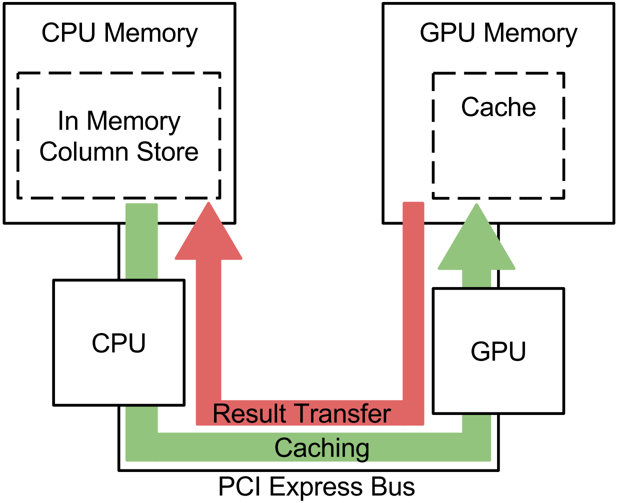
We can either

- Decide on an operator placement, and transfer the input data accordingly, or

- Decide on a data placement, and place the operators accordingly

S. Breß. H. Funke, J. Teubner. Robust Query Processing in Co-Processor-accelerated Databases. In SIGMOD, 2016.

Classical Operator-Driven Data Placement



Data-Placement-Driven Operator Placement

Key Observations:

1. We do not have to use a co-processor, if we expect high overheads for data transfer
$\rightarrow$ Process data on CPU, and use co-processor only when advantageous (e.g., input data cached)

2. If the input data for an operator is cached on the co-processor, the co-processor is (most of the time) faster than the CPU
$\rightarrow$ Push operators (code) to data

S. Breß. H. Funke, J. Teubner. Robust Query Processing in Co-Processor-accelerated Databases. In SIGMOD, 2016.

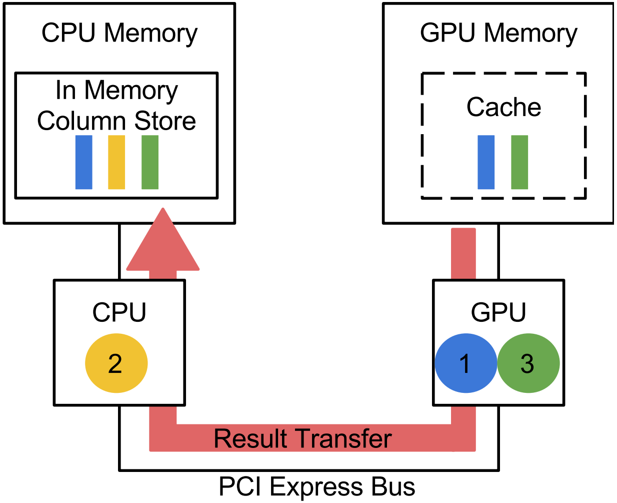
Data-Placement-Driven Operator Placement

Algorithm consists of two stages:

1. Periodically run a data placement thread in the background:
   - Identify part of the working set that fits in co-processors data cache
     $\rightarrow$ Use most recently or most frequently used columns/data pages
   - Transfer the selected columns/data pages to the co-processors cache
2. For each operator:
   - Test whether input data is cached on the co-processor (this includes result data from previous operators)
   - If the data is cached, execute operator on co-processor, if not, execute operator on CPU

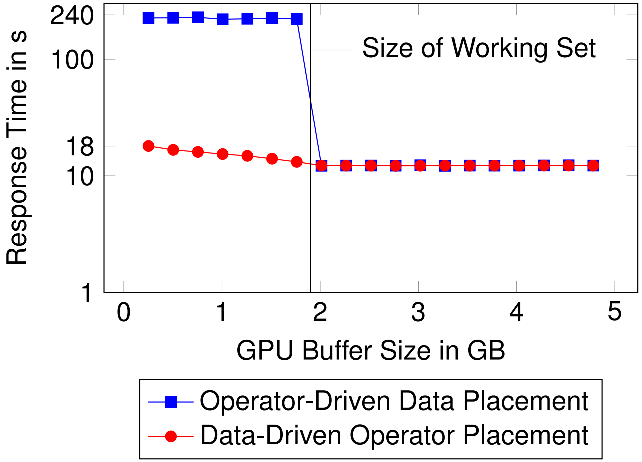
S. Breß. H. Funke, J. Teubner. Robust Query Processing in Co-Processor-accelerated Databases. In SIGMOD, 2016.

Data-Placement-Driven Operator Placement



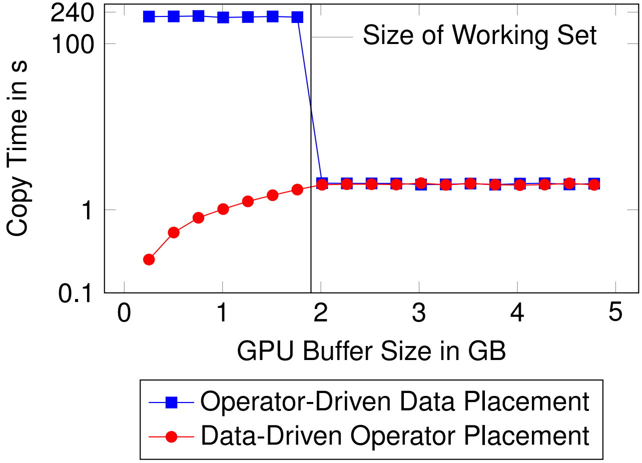
S. Breß. H. Funke, J. Teubner. Robust Query Processing in Co-Processor-accelerated Databases. In SIGMOD, 2016.

Data to Query or Query to Data?



Data-driven operator placement solves the cache thrashing problem

Data to Query or Query to Data? (2)



Summary

Cache Thrashing

- Problem: Scarce cache memory leads to expensive eviction and recaching and slows down query processing
- Solution: Data-centric operator placement, which pushes operators to processors that have data cached

S. Breß. H. Funke, J. Teubner. Robust Query Processing in Co-Processor-accelerated Databases. In SIGMOD, 2016.

CoGa
DB

http://cogadb.dfki.de

Outline

Summary

Hardware trends

- Increasing heterogeneity: CPUs, GPUs, co-processors
- Many-core processors, even on CPUs
- Disk IO is not the limiting factor anymore

Consequences

- Need to actively adjust the DBMS to make use of heterogeneous processors (e.g., relational operators, query processor)
- Requires to rethink architectural aspects of the DBMS (e.g., columns stores, compression)

Processors are parallel and heterogenenous, we have to deal with it!

References I

[1] S. Borkar and A. A. Chien.
The future of microprocessors.
Communications of the ACM, 54(5):67–77, 2011.

[2] J. Casper and K. Olukotun.
Hardware acceleration of database operations.
In *FPGA*, pages 151–160. ACM, 2014.

[3] R. Dennard, V. Rideout, E. Bassous, and A. LeBlanc.
Design of ion-implanted mosfet's with very small physical dimensions.
IEEE Journal of Solid-State Circuits, 9(5):256–268, 1974.

[4] H. Esmaeilzadeh, E. Blem, R. St. Amant, K. Sankaralingam, and D. Burger.
Dark silicon and the end of multicore scaling.
In *ISCA*, pages 365–376. ACM, 2011.

[5] N. Hardavellas, M. Ferdman, B. Falsafi, and A. Ailamaki.
Toward dark silicon in servers.
IEEE Micro, 31(4):6–15, 2011.

[6] B. He, M. Lu, K. Yang, R. Fang, N. K. Govindaraju, Q. Luo, and P. V. Sander.
Relational query co-processing on graphics processors.
In *ACM Trans. Database Syst.*, volume 34. ACM, 2009.

References II

[7] J. He, M. Lu, and B. He.
Revisiting co-processing for hash joins on the coupled CPU-GPU architecture.
Proc. VLDB Endow., 6(10):889–900, 2013.

[8] J. He, S. Zhang, and B. He.
In-cache query co-processing on coupled CPU-GPU architectures.
Proc. VLDB Endow., 8(4):329–340, 2014.

[9] J. L. Hennessy and D. A. Patterson.
Computer Architecture: A Quantitative Approach.
Morgan Kaufmann Publishers Inc., 5th edition, 2011.

[10] S. Jha, B. He, M. Lu, X. Cheng, and H. P. Huynh.
Improving main memory hash joins on intel xeon phi processors: An experimental approach.
PVLDB, 8(6):642–653, 2015.

[11] M. Lu, L. Zhang, H. P. Huynh, Z. Ong, Y. Liang, B. He, R. Goh, and R. Huynh.
Optimizing the mapreduce framework on Intel Xeon Phi coprocessor.
In *Big Data*, pages 125–130. IEEE, 2013.

[12] G. Moore.
Cramming more components onto integrated circuits.
Electronics, 38(8), 1965.

References III

[13] R. Mueller, J. Teubner, and G. Alonso.
Data processing on FPGAs.
PVLDB, 2(1):910–921, 2009.

[14] K. Sattler, J. Teubner, F. Beier, and S. Breß.
Many-core-architekturen zur datenbankbeschleunigung.
In *Datenbanksysteme für Business, Technologie und Web (BTW 2015) - Workshopband, 2.-3. März 2015, Hamburg, Germany*, pages 269–270, 2015.