Co-Processor Accelerated Data Management

Sebastian Breß
TU Dortmund
DBIS Group

sebastian.bress@cs.tu-dortmund.de

Summer Term 2015
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 [19] (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 [16])
  - Power density of transistors was constant
    → Smaller transistors required less voltage and current (Dennard scaling [4])
- 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

→ Constant chip size and an increasing number of transistors increases overall power consumption and the produced heat [12]
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
  → 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) [5].
- Experts predict that this trend will not scale beyond a few hundred cores [7].
- 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%! [5]
Motivation

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 → \textit{dark silicon} [1, 5, 7]

How can we exploit the increasing number of transistors?

- Provide a large number of cores that are \textit{specialized} for certain tasks
- Cores that are most suitable for the current workload are powered on until the energy budget is reached [5, 7]
- Inevitably, this will cause processors to become increasingly \textit{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, 7]

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 ➔ !!!

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 [8]
  - APUs [9, 10]
  - MICs [13, 14]
  - FPGAs [3, 17]
- We need to think about how we can exploit all processors during query processing.
Outline

Motivation

GPU Architecture

Programming GPUs

Query Processing

Heterogeneous Platforms

Architecture of Co-Processor-Accelerated DBMSs

Lessons Learned
GPGPU for DBMS?

- tremendous computation power & massive parallelism

<table>
<thead>
<tr>
<th>Processor</th>
<th>FLOPS</th>
<th>Cores</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intel Core i7 (Sandy Bridge)</td>
<td>≈ 102 GFLOPS</td>
<td>4</td>
</tr>
<tr>
<td>Tesla K40 (Kepler)</td>
<td>≈ 1.66 TFLOPS</td>
<td>2880</td>
</tr>
</tbody>
</table>

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.

→ geometry shaders idle for pixel-heavy workloads and vice versa

→ unified model with general-purpose cores

Thus: Design inspired by CPUs, but different

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

PCI Express bus

CPU

Memory

GPU

Memory

S. Breß
Example: NVIDIA GPUs

NVIDIA GTX 280

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

Source: www.hardwaresecrets.com
Inside a Streaming Multiprocessor

- **I-Cache**
- **MT Issue**
- **C-Cache**
- **SP**
- **SP**
- **SP**
- **SP**
- **SFU**
- **SFU**
- **DP**
- **shared memory**

- 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
  \[ \rightarrow 30,720 \text{ concurrent threads} (!) \]
Inside a Streaming Multiprocessor: NVIDIA Fermi

- 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

- **Register**
- **L1 Shared Memory**
- **Stream Multiprocessor**

**L2**
- **Global Memory**
- **Host Memory**
- **Mapped**

- **8.000 GB/s**
- **177 GB/s**
- **8 GB/s**
- **16...48 KB**
- **1.600 GB/s**
GPU: Execution Model

- data copying host (CPU) ↔ 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.
  → Large thread pool rather than caches
    (This idea is similar to SMT in commodity CPUs)

▶ Assume data parallelism and restrict synchronization.
  → Threads and small groups of threads use local memories.
  → Synchronization only within those groups (more later).

▶ Hardware thread scheduling (simple, in-order).
  → Schedule threads in batches (≈ “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.

- For memory access warps are split into **half-warps** consisting of 16 threads.
- 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

- **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

\(^a\) no dual issue for double-precision instr.
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**.

→ **Hide latencies** by executing other warps when one is paused.
→ Need **enough warps** to fully hide latency.

e.g.,

- Need \( \frac{24}{4} = 6 \) warps to hide register dependency latency.
- Need \( \frac{400}{4} = 100 \) instructions to hide memory access latency. If every 8th instruction is a memory access, \( \frac{100}{8} \approx 13 \) 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
  - 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 \( \Rightarrow \) 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
- \( \ldots \) and share the L2 cache
Outline

Motivation

GPU Architecture

Programming GPUs

Query Processing

Heterogeneous Platforms

Architecture of Co-Processor-Accelerated DBMSs

Lessons Learned
GPU Programming: Basic Concepts

- GPUs are separate devices connected via a high-speed interface to the host computer
  - GPU runs in a separate memory space
    - requires memory transfer between host and device
    - GPU memory bandwidth: 160 \ldots 200 \text{ GB/s}
    - host memory bandwidth: 8 \ldots 20 \text{ GB/s}
    - CUDA’s Unified Virtual Addressing (UVA) hides the details of transfer but does not close the bandwidth gap

- Basic unit of work on a GPU is a thread
  - GPU programs utilize kernels
    - kernel = subroutine executed on the GPU and is callable from the host
    - kernels are called asynchronously
    - kernel calls are queued by the host for execution
    - host does not wait for finishing
CUDA

- NVIDIA’s Compute Unified Device Architecture = programming model and compute platform
- only available for NVIDIA (GP)GPUs
- SDK provides
  - CUDA compiler driver `nvcc`
  - low level API (C)
  - higher level APIs + libraries
CUDA Toolchain

```
.gpp

.cpp

g++

Host.o

nvcc

Host functions

CUDA kernels

g++

host.o

cudacc

CUDA.o

Executable/.so

ld

S. Breß
Co-Processor Accelerated Data Management
30 / 126
```
CUDA: A Simple Kernel

```c
__global__ void simple(void) {}

int main(void) {
    simple<<<1, 1>>();
}
```
CUDA Kernels

- `kernel<<<N, T>>>`
  - launch N copies of kernel
    - use `blockIdx.x` to access block index
  - launch kernel with N threads
    - use `threadIdx.x` to access thread index
CUDA: Blocks vs. Threads

- kernel launches a **grid** of **thread blocks**
- thread block = batch of threads
  - within a block threads can cooperate via shared memory and synchronize their execution
  - no cooperation between threads from different blocks
- **multidimensional IDs** simplify addressing of multidimensional data:
  - block ID: 1D, 2D
  - thread ID: 1D, 2D, 3D
CUDA Kernels

- $M = 6$ threads/block

```plaintext
threadIdx.x

0 1 2 3 4 5 0 1 2 3 4 5 0 1 ...
```

```
blockIdx.x = 0    blockIdx.x = 1
```

```plaintext
index = threadIdx.x + blockIdx.x \times M
```
CUDA: Example

```c
__global__ void addVecKernel(int *vec, int *res, int C) {
    int tid = threadIdx.x + blockIdx.x * blockDim.x;
    res[tid] = vec[tid] + C;
}

int *d_vec, *d_res;
cudaMalloc((void **)d_vec, sizeof(int) * N_ELEMS);
cudaMalloc((void **)d_res, sizeof(int) * N_ELEMS);
cudaMemcpy(d_vec, &vec, sizeof(int) * N_ELEMS, cudaMemcpyHostToDevice);
addVecKernel<<<10, 100>>>(d_vec, d_res, C);
```
Thrust: CUDA C++ Template Library

- based on STL
- provides data parallel primitives (scan, sort, reduce) and vector containers for host and device memory

```cpp
thrust::host_vector<int> h_vec(N_ELEMS);
thrust::device_vector<int> d_vec(N_ELEMS), res(N_ELEMS);

// transfer from host to device
thrust::copy(h_vec.begin(), h_vec.end(), d_vec.begin());

// kernel with lambda expression
thrust::transform(d_vec.begin(), d_vec.end(), res.begin(), _1 + C);
```
Higher-Level-APIs

- **OpenACC**: directive-based approach similar to OpenMP, but for heterogeneous platforms
- **OpenCL-based**
  - clBLAS: basic linear algebra
  - clFFT: Fast Fourier transformation
  - VexCL: vector expression
  - boost.compute: Boost/STL-like wrapper over OpenCL
- **CUDA-based (GPU only)**
  - cuBLAS
  - cuFFT
  - ArrayFire: array-based processing
Three Rules of GPU Programming


1. Get the data on the GPU and keep it there.
2. Give the GPU enough work to do.
3. Reuse data within the GPU to avoid memory bandwidth limitations.
Outline

Motivation

GPU Architecture

Programming GPUs

Query Processing

Heterogeneous Platforms

Architecture of Co-Processor-Accelerated DBMSs

Lessons Learned
Co-processing in a DBMS

- Query Processing
  - Relational
    - Selection
    - Projection
    - Join
  - Other
    - Online Aggregation
    - Sorting
    - Map Reduce
- Query Optimization
  - Selectivity Estimation
- Searching
  - Index Lookups
  - knn-Search
  - Range Queries
- XML
  - XPath Selection
- Database Tasks
  - Compression
  - Update Merging in Column Stores
  - Transaction Management
Selection

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

**pred: val<5**

```
val | res
---|---
1   | 1
5   | 3
3   | 2
7   |
2   |
4   |

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
  → Ensure correctness by latches

- Latching may serialize threads and nullify the performance gained by parallel execution
  → Need to ensure correctness without latching

- Key Idea: Pre-compute write locations

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:
  \[ R_{out}[i] = R_{in}[0] + \ldots + R_{in}[i - 1] \quad (1 \leq i < |R_{in}|) \]
  \[ R_{out}[0] = 0 \]

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
  - $flags$ array contains a 1 if the corresponding tuple in $R$ is part of the result
  - 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 $ps$
  - Now we have the write locations for each tuple in the result buffer

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:

```c
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];
    }
}
```

Parallel Filter: Example

- **val:**
  - 1
  - 5
  - 3
  - 7
  - 2
  - 4

- **flags:**
  - 1
  - 0
  - 1
  - 0
  - 1
  - 1

- **ps:**
  - 0
  - 1
  - 1
  - 2
  - 2
  - 3

- **res:**
  - 1
  - 3
  - 2
  - 4

**Build flag array:**
- if\(\text{pred}(\text{val}[i])\)
  - \(\text{flags}[i]=1;\)
- else
  - \(\text{flags}[i]=0;\)

**Compute prefix sum from flags:**

**Scan flags and write \(\text{val}[i]\) to position \(\text{ps}[i]\) in result array:**
Joins

Selections $\sigma(R)$:
- Tuples either match or not
- Maximal size of result is as large as the input relation

Joins $R \bowtie S$:
- Tuples can match more than once
- Maximal size of join result can be very large ($|R| \cdot |S|$))
Joins

General Problems:

- Exact result size not known in advance
  (exception: primary-key/foreign-key join)

- Join result may or may not fit in GPU memory

- Need lock-free processing to fully exploit parallelism of GPU
  → Pre-compute write locations for each thread
Joins

All joins use a three-step output scheme (Exception: PK-FK Joins):

1. Each thread counts the number of join partners for its share of the input

2. Using the result size for each thread, we compute a prefix sum, to get the write location for each thread

3. The host allocates the memory of the size of the join result and all threads write their results to the device memory according to their write locations

→ lock free processing scheme

[8]
Example: Indexed Nested Loop Join

Three steps ($R \bowtie S$):

1. Build index on smaller relation $R$ by sorting
2. Join: Lookup each element of $S$ in sorted array by binary search
3. Transform the result positionlist of $R$ to index original unsorted $R$

- Remember: We have to perform the second step two times!
- Step 1 and 3 can be omitted in case $R$ is sorted
Example

- Result of joins in column stores consist of two position lists $P_R$ and $P_S$
- Positions on the same offset in $P_R$ and $P_S$ describe two matching tuples
  - $P_R$ and $P_S$ always have the same number of elements
Building Index On The Fly

- We need to store a position list that maps to the original order
- We require this position list for step 3!
Indexed Nested Loop Join: First Phase

- Each thread accesses one element and performs a lookup on the index.
- Threads increase their read offset by the total number of threads until becomes larger than $|S|$.
- Each thread counts the number of matches and stores it in counter.
  - More threads increase memory consumption.
- Compute prefix sum of counter array to get thread write positions and result size.
Indexed Nested Loop Join: Second Phase

- Threads lookup their assigned elements and use the prefix sum array to write the result.
- If we had to sort \( R \) first, then we need to reverse the sorting.
Query Processing

Transform Result Position List of Sorted Relation

Step 1: Sorting

<table>
<thead>
<tr>
<th>R</th>
<th>T</th>
<th>R'</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>5</td>
</tr>
</tbody>
</table>

Position List
Sorted Array

Step 3: Revert Sorting

<table>
<thead>
<tr>
<th>T</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
</tr>
</tbody>
</table>

Gather

Perform a gather operation on position list from step 1 (T) using PR:

Gather Operation \( R_{out} = \text{gather}(R_{in}, P) \):

- Input: Array \( R_{in} \), Position List \( P \)
- Output: Array \( R_{out} \) (\( R_{out} \) has as many elements as \( P \))
- Operation: \( R_{out}[i] = R_{in}[P[i]] \), with \( i \in \{0, \ldots, |P| - 1\} \)
Example: Nested Loop Join Count Kernel of CoGaDB

template<typename T>
__global__ void join_and_count_kernel(T* column_left, size_t count_column_left,  
    T* column_right, size_t count_column_right, TID* times_found){
    TID tid = threadIdx.x + blockIdx.x * blockDim.x;
    while(tid<count_column_left){
        TID index = binary_search_first_occurrence(column_right,  
            count_column_right, column_left[tid]);
        TID count = 0;
        if(index < count_column_right) {
            T value = column_right[index];
            count++;
            while(index+count < count_column_right) {
                if(column_right[index+count] == value) {
                    count++;
                } else break;
            }
        }
        times_found[tid] = count;
        tid += blockDim.x * gridDim.x;
    }
} //taken from gpudbms/cogadb/src/backends/gpu/gpu_join.cu
GPU-assisted Query Optimization

- Key Idea: Use massive parallelism of GPUs and other co-processors to produce better plans in the same time

- Applicable to all database systems

- Example: Selectivity estimation using kernel density estimators (KDE)
GPU as Statistical Co-Processor

Max Heimel, Volker Markl, A First Step Towards GPU-Assisted Query Optimization. ADMS@VLDB, 2014.
Kernel Density Estimator

Max Heimel, Volker Markl, A First Step Towards GPU-Assisted Query Optimization. ADMS@VLDB, 2014.
GPU Parallelization Strategy

Query Information
(used by all threads)

from Host

first data point

... Data sample

Computing local contributions
(one thread per data point)

... Local contributions

Parallel aggregation

Final Result

to Host

Max Heimel, Volker Markl, A First Step Towards GPU-Assisted Query Optimization. ADMS@VLDB, 2014.
Runtime: CPU vs. GPU

Max Heimel, Volker Markl, A First Step Towards GPU-Assisted Query Optimization. ADMS@VLDB, 2014.
Impact of Correlation on Estimation Quality

Max Heimel, Volker Markl, A First Step Towards GPU-Assisted Query Optimization. ADMS@VLDB, 2014.
Outline

Motivation

GPU Architecture

Programming GPUs

Query Processing

Heterogeneous Platforms

Architecture of Co-Processor-Accelerated DBMSs

Lessons Learned
Heterogeneous Platforms

Motivation

PCI Express Bus

S. Breß

Co-Processor Accelerated Data Management
Heterogeneous Platforms

Motivation

PCI Express Bus

S. Breß
How can we accelerate database query processing when we have multiple heterogeneous processors?
We should care about where the data is!

![Bar chart showing execution times for different memory types.]

- CPU Only: 1.0 seconds
- GPU Cold Memory: 3.1 seconds
- GPU Hot Memory: 0.4 seconds
State of the Art

- Most publications investigated single database operators on processors in isolation

- Little attention on coupled CPU/Coprocessor processing
Research Challenges

1. Choose processor for an operator (operator placement)

2. Process multiple *operators* concurrently

3. Process multiple *queries* concurrently
Operator Placement

Query Plan

Placed Query Plan

CPU

GPU
Operator Placement

- Schedule each operator in a query plan on a processor

- For $n$ processors and $m$ operators, query optimization needs to explore $n^m$ query plans

- Requires different cost modeling compared to regular DBMS
  → Estimate execution times for operator placement
State of the Art: Run-Time Estimation

- Most approaches use analytical cost models: predictable, fast

- Hardware becomes more heterogeneous
  → Analytical cost models do not scale
Analytical Cost Models: Parameters

Each cost model considers:

- Features of data (e.g., size, selectivity)
Analytical Cost Models: Parameters

Each cost model considers:

- Features of data (e.g., size, selectivity)

- Hardware dependent parameters (e.g., cores, clock rate, memory bandwidth)
Analytical Cost Models: Parameters

Each cost model considers:

- Features of data (e.g., size, selectivity)

- Hardware dependent parameters (e.g., cores, clock rate, memory bandwidth)

- The algorithms implementation details
Analytical Cost Models: Parameters

Each cost model considers:

- Features of data (e.g., size, selectivity)
- Hardware dependent parameters (e.g., cores, clock rate, memory bandwidth)
- The algorithms implementation details
- Data transfer cost
Analytical Cost Models: Parameters

Each cost model considers:

- Features of data (e.g., size, selectivity)
- Hardware dependent parameters (e.g., cores, clock rate, memory bandwidth)
- The algorithms implementation details
- Data transfer cost
- Current load condition on processors
Scalability Problem

- DBMS needs to maintain one cost model for each:
  - Database algorithm
  - Processor type and generation

- Vendor needs to maintain cost models
  → Increased maintenance cost
Scalability Problem (2): Example

- #Database algorithms: 10
- #Processor types: 3 (CPU, GPU, MIC) → Growing!
- #Processor generations: 3

→ Vendor maintains $10 \cdot 3 \cdot 3 = 90$ cost models?
→ He won’t...
Second Try: Learned Cost Models

- Learning approaches work independent of processors
  → Hardware Oblivious

- Require major cost factors to be modeled as feature vector
  + Automatically adapt to changing data, database algorithms, and workloads

- Slower computation of estimations
  → Can we manage the overhead?
Architecture of Hardware Oblivious Query Optimizer

- Learning Cost Estimator
- Algorithm Selector
- Query Optimizer
- hybrid CPU/CP DBMS
- HyPE Components
- DBMS Adapter

Hybride Inter-Operator Ansätze
Decision Model for Operator Placement

- Algorithm as central component of abstraction
- Learn execution behavior of algorithms
- Pick algorithm that is likely to be the fastest for execution
Decision Model for Operator Placement

- Algorithm as central component of abstraction
- Learn execution behavior of algorithms
- Pick algorithm that is likely to be the fastest for execution
Impact of Runtime Adaptation on Performance

![Graph showing impact of runtime adaptation on performance with curves for Q1.1, Q2.1, Q3.1, and Q4.1 over iterations.](attachment:image.png)
Load Tracking

Often one processor outperforms the others for a certain operator:

- Potential overload of certain processors

- Other processors may become underutilized or idle

Goal: Exploit inter-processor parallelism to improve performance
Load Tracking

Operator to Place | Ready Queues | Processing Devices
---|---|---
Unplaced Operator | Placed Operator | Running Operator

S. Breß
Load Tracking

Operator to Place

Ready Queues

Processing Devices

Unplaced Operator

Placed Operator

Running Operator

Minimal Estimated Execution Time
Load Tracking: Performance Improvements

![Bar chart showing speedup w.r.t. fastest processor for different operations: Aggregation (1.7), Join (1.3), Selection (1.2), Sort (1.4).]
Ressource Limitations

- Co-Processor’s memory capacity quite small

- Cannot fit all data on the co-processor

- But what about a lossily approximation?

Approximate & Refine

**Approximation:**
- Operates on approximated data on co-processor
- Returns a superset of the result

**Refinement:**
- Operates on approximated and refined data on CPU
- Eliminates false positives from approximated output

**Compute Min/Max with A & R Framework**

### Diagram
- **Approximation**: The diagram illustrates the trade-off between accuracy and efficiency in computing min/max values.
- **Correct Minimum**
- **False Minimum**
- **Condition**
- **Approximate Condition**

### Queries
- **Precise Query**: `select min(Y) from R where x>6`
- **Approx. Query**: `select min(Y) from R where x>=4`

---

Example A & R Query Plan

Performance for Range Queries

Outline

Motivation

GPU Architecture

Programming GPUs

Query Processing

Heterogeneous Platforms

Architecture of Co-Processor-Accelerated DBMSs

Lessons Learned
Literature

Most of the content of this topic can be found in the following publication:

Row-Wise vs. Column-Wise Storage

The $n$-ary storage model (NSM, row-wise storage) is not the only choice.

Column-wise storage (decomposition storage model, DSM):
Row Stores vs. Column Stores

A column store is more suitable for a GPU-accelerated DBMS than a row store.

Column stores:

▶ Allow for coalesced memory access on the GPU
▶ Achieve higher compression rates, an important property considering the current memory limitations of GPUs
▶ Reduce the volume of data that needs to be transferred to GPU RAM
Processing Model

There are basically two alternative processing models that are used in modern DBMS:

- Tuple-at-a-time volcano model \[^6\]
  - Operator requests next tuple, processes it, and passes it to the next operator

- Operator-at-a-time bulk processing \[^{15}\]
  - Operator consumes its input and materializes its output
Tuple-at-a-time processing

Advantages:
- Intermediate results are very small
- Pipelining parallelism
- The "classic" approach

Disadvantages:
- A higher per tuple processing overhead
- High miss rate in the instruction cache
Operator-at-a-time Processing

Advantages:

- Cache friendly memory access patterns
  → Making effective usage of the memory hierarchy \(^{[15]}\)
- Parallelism inside an operator, multiple cores used for processing a single operation
  → Intra-operator parallelism

Disadvantages:

- Increased memory requirement, since intermediate results are materialized \(^{[15]}\)
- No pipeline parallelism
Processing Model for GPU-aware DBMS

Operator-at-a-time processing is more promising than tuple-at-a-time processing, because:

- Data can be most efficiently transferred over the PCIe bus by using large memory chunks
- Tuple-wise processing is not possible on the GPU, because inter-kernel communication is undefined \(^\text{[18]}\) → No pipelining possible
- Operator-at-a-time processing can be easily combined with operator-wise scheduling
Cross-Device Optimizations

- Avoid data transfers
  - Data placement strategies
  - GPU-aware optimizer
- Reduce cost of data transfers
  - Compression
  - Overlap transfer with processing
  - Use pinned host memory
Caching and Data Placement

- CPU Memory (64 GB)
  - In Memory Column Store
- GPU Memory (5 GB)
  - Cache (2.5 GB)
- Result Transfer
- Caching
- PCI Express Bus
Caching and Data Placement (2)

Caching:
- Copying data to co-processors is very expensive
- Common to hide high communication latencies by caches
  → Use part of the co-processor memory to cache frequently accessed data

Data Placement Strategy:
- Choose which part of the database is kept in the cache
- Related to classical page replacement strategies in disk-based databases
  → But: We can process data in main memory as well, we do not have to copy data to the co-processor if we do not use it
Data Compression

Problems with Caching:

- We can not always avoid data transfers between processors
  → We have to at least copy result back to main memory
- Dedicated memory of co-processors has little capacity

Reduce the impact of the communication bottleneck:

- Reduce the performance penalty by compressing the data
  → Copying smaller chunks is faster
  - Keep more data cached
  - Query processing on compressed data is often faster than on uncompressed data
Pinned Host Memory

- On default, all memory pages are pagable
  - Operating system is free to move page to swap on disk

- CPU has better things to do then performing memory transfers to the GPU
- Data transfer is the job of the Direct Memory Access (DMA) controller
- DMA copies memory independent from the CPU

What happens when the DMA controller tries to copy a memory page that was just replaced by another page?
Pinned Host Memory (2)

- DMA controller can only transfer data from main memory to GPU memory when memory pages are pinned (not pageable)
- Data needs to be copied from pageable memory pages to pinned memory pages
- Expensive
Impact of Pinned Memory on Data Transfer Performance

Figure: Bandwidth of Copy Operations from CPU to GPU for 16 MB of data.

CPU: Intel(R) Core(TM) i7-4770, GPU: GeForce GTX 660
Data Transfers from Pageable Memory

GPU Memory

CPU Memory

Pageable Memory

Pinned Memory
Keep Data in Pinned Host Memory?

- Frequently accessed data can be kept in pinned memory
  - Avoid host side copy operation

Why can we not keep the whole database in pinned host memory?

- Pinned memory bypasses the virtual memory management of the OS
- The more pinned memory we allocate, the less flexibility is left for the OS
  - Memory allocation request will fail much sooner for pinned memory than for pageable memory
  - We have to use pinned host memory moderately
Overlap of Data Transfer and Processing

- Data transfers between CPU and GPU are expensive
- High latency until GPU can start processing
→ Hide latency of data transfers by overlapping them with computations
Overlap of Data Transfer and Processing (2)

Copy Input from CPU to GPU
Perform Computation
Copy Result GPU to CPU

- Different colors indicate independent streams that can process data independent of each other
- Modern GPUs can also transfer data from CPU to GPU and GPU to CPU concurrently
  \[\Rightarrow\] We can overlap input transfer, computation, and result transfer
Device-Dependent Optimizations

- Processing Models
  - Block at a time
    - Query Compilation
  - Operator at a time
    - All in one kernel
    - Parallel Primitives
Block-oriented Query Processing

- Until now, we assumed operator-at-a-time processing
- By materializing the result, it is pushed out of the cache and is written back to main memory
- Next operator reads the data again

→ Overhead

Can we avoid these additional read and write operations?
Block-oriented Query Processing

Taken From: Figure 3 from [20]

- "tuple at a time" for DBMS "X"
- MySQL 4.1
- "column at a time" for MonetDB/MIL
- "vector at a time" for MonetDB/X100

- Hand-Coded C Program
- Interpretation overhead decreases
- "low interpretation overhead in-cache materialization"
- "main-memory materialization overhead"
- "query without selection"
- "vectors start to exceed CPU cache, causing extra memory traffic"
Block-oriented Query Processing

- Process small blocks (vectors) that fit into the cache (or shared memory)
- Execute all operators on the cache resident vector
- Results have to be materialized only for pipeline breakers
  - Improved efficiency

Also a cross-device optimization:
- Stream vectors from CPU to GPU or vice versa
- Overlap transfer and compute operations of different vectors
Portability

- Up to now, we talked about performance
- What about implementation effort for co-processor accelerated databases?

The classic way: Hardware-Sensitive Database Operators

- Implement each database operator on all supported processors (i.e., on CPU and GPU)
- Allows for performance tuning specific to each processor
- High implementation effort

How can we support multiple heterogeneous processors without rewriting database operators for each processor?
Hardware Oblivious Database Operators

![Diagram showing the architecture of co-processor accelerated DBMSs]

- **Main Codebase**
  - Device-Specific Operators
  - Device-Specific Operators

- **Engine Code**
  - Main Codebase
  - Hardware-oblivious Operators
  - Parallel Programming Library

- **Compiler**
- **Binary**
- **Driver**

- **CPU**
- **GPU**
- **GPU 1**
- **GPU 2**

Taken From: Figure 1 From [11]
Hardware Oblivious Database Operators

- Write one set of hardware-oblivious operators, written against a parallel programming library.
- These operators are compiled to a generic binary format that is later mapped to the hardware by a so-called driver.
- Drivers translate code to processor-specific instructions and perform processor-specific optimizations.
- Performance is not as good as with the hardware-sensitive approach, but achieves reasonable performance for a small fraction of the implementation cost.
Reference Architecture for a GPU-accelerated DBMSs

- SQL and Logical Optimization
- Physical Optimizer
  - Cost Model
  - CPU/GPU Scheduler
  - Hybrid Query Optimizer
- Relational Operators
- Access methods
- Data parallel primitives (e.g., map)
- Data Placement Strategy
- In-memory storage (columnar, compressed)

Logical query plan

Hybrid query plan calls relational operators

Implemented on top of

Calls primitives

Process input data

Fetch input data and copy to device memory

Existent in traditional main-memory DBMSs

Main-memory DBMSs extension for GDBMSs

New component in GPU-accelerated DBMSs
Outline

Motivation

GPU Architecture

Programming GPUs

Query Processing

Heterogeneous Platforms

Architecture of Co-Processor-Accelerated DBMSs

Lessons Learned
Observations

Hardware trends

- increasing heterogeneity: CPUs, GPUs, co-processors
- many-core processors, even on CPUs
- disk IO isn’t the limiting factor anymore

Complex database tasks

- scanning large amount of in-memory structures
- integration of compute-intensive tasks: statistical functions, complex data, . . .

The world is parallel, we have to deal with it!
Database Programming for Many-Core Architectures

- **Parallel programming for shared memory**
  - partitioning of data for exploiting data parallelism
  - minimize synchronization/coordination overhead
  - try to keep the cores busy

- Choose an **appropriate programming model/API**
  - increase productivity/reduce LOC
  - increase portability of code

- **Optimize code** for specific hardware
  - Cache awareness, access patterns
  - vectorization, SIMD, alignment, bandwidth ↔ latency, …
Consequences

- Parallelization is important!
  - Hardware development
  - Amdahl’s Law
  
  \[ S = \frac{1}{(1-P) + \frac{P}{N}} \]
  
  - Speedup
  - \( P \): Proportion of parallel code
  - \( N \): Number of processors

- Performance limited by sequential program parts!

- Think parallel!
  - Work with parallel hardware teaches a lot of things.
  - Parallel algorithmic patterns
  - Behavior of underlying architecture
  - Optimization tricks

http://en.wikipedia.org/wiki/Amdahls_law
Pitfalls

▶ Beware of overheads!
  ▶ Enough work per thread. . .
  ▶ . . . but not too much
▶ Know the strengths and weaknesses of the underlying hardware!
  ▶ Sometimes brute force better than elaborate algorithms
  ▶ Hybrid approaches can be beneficial
  ▶ Sometimes good engineering better than deep research

H. Kaiser

Plain Threads are the GOTO of Today's Computing.

Keynote@Meeting C++, 2014.
GPGPUs in DBMS?

GPGPUs …

- can be used to speed up compute-intensive tasks (co-processing)
- are a great playground for exercising and teaching massive parallel programming
- have some **pitfalls** which have to be taken into account: data transfer, SIMT processing, warp divergence
References I

The future of microprocessors.

GPU-accelerated database systems: Survey and open challenges.

Hardware acceleration of database operations.

Design of ion-implanted mosfet’s with very small physical dimensions.

Dark silicon and the end of multicore scaling.

Encapsulation of parallelism in the volcano query processing system.
References II

Toward dark silicon in servers.

Relational query co-processing on graphics processors.

Revisiting co-processing for hash joins on the coupled CPU-GPU architecture.

In-cache query co-processing on coupled CPU-GPU architectures.

Hardware-oblivious parallelism for in-memory column-stores.

References III

Improving main memory hash joins on intel xeon phi processors: An experimental approach.

Optimizing the mapreduce framework on Intel Xeon Phi coprocessor.

Database architecture evolution: Mammals flourished long before dinosaurs became extinct.

Cramming more components onto integrated circuits.
Electronics, 38(8), 1965.

Data processing on FPGAs.

[18] NVIDIA.
NVIDIA CUDA C Programming Guide.
Many-core-architekturen zur datenbankbeschleunigung.

Monetdb/x100 - a dbms in the cpu cache.