technische universität dortmund

GPU-accelerated Data Management

Data Processing on Modern Hardware

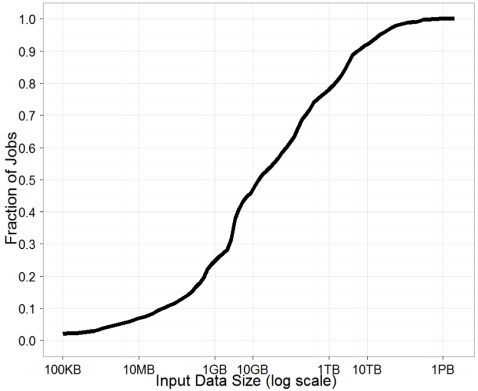
Sebastian Breß
TU Dortmund University
Databases and Information Systems Group

Summer Term 2014

Motivation – Big Picture

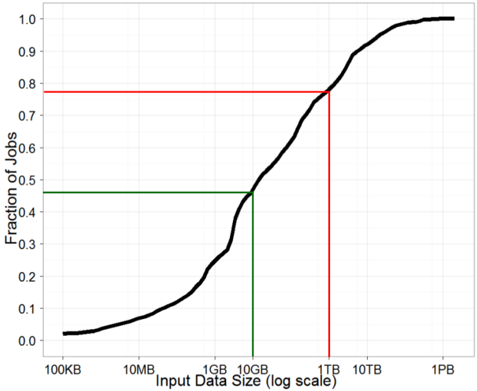
- **Big Data Analysis:** Should we always scale out, e.g., use the cloud to analyze our data?

Motivation – Big Picture



Taken from [Appuswamy et al., 2013]

Motivation – Big Picture



Taken from [Appuswamy et al., 2013]

- $\approx$ 50% of the job sizes are smaller than 10 GB
- $\approx$ 80% of the job sizes are smaller than 1 TB

$\rightarrow$ A majority of big data analysis jobs can be processed in one scale up machine!

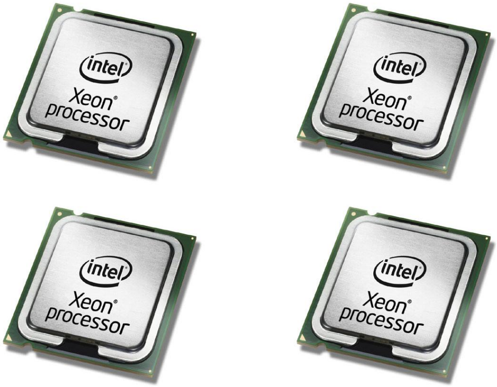
[Appuswamy et al., 2013]

Motivation – Big Picture

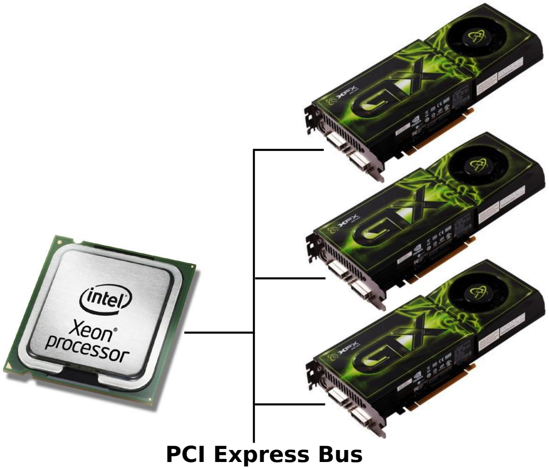
How to scale up a server?

Base Configuration

Scale Up: Add More CPUs



Scale Up: Add GPUs



Focus of this Lecture Topic

How can we speed up database query processing using GPUs?

Graphics Processing Unit: Architecture

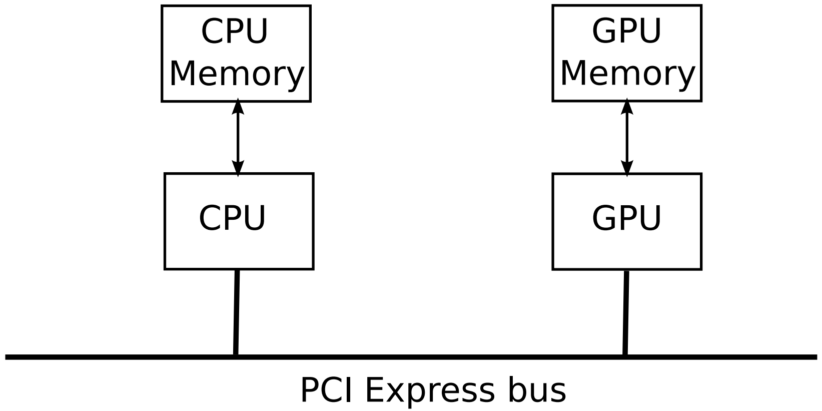
Recapitulation: The Central Processing Unit (CPU)

- General purpose processor
- Goal is low response time:
  $\rightarrow$ optimized to execute <u>one</u> task as fast as possible
  (pipelining, branch prediction)
- Processes data dormant in the main memory

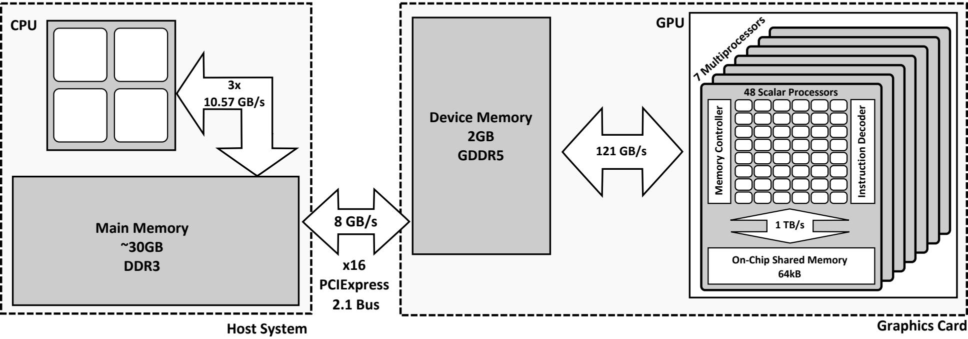
Graphics Processing Unit (1)

- Specialized processor, can be programmed similar to CPUs
- GPUs achieve high performance through massive parallelism
  $\rightarrow$ Problem should be easy to parallelize to gain most from running on the GPU
- Single Instruction, Multiple Data (SIMD): Each multiprocessor only has a single instruction decoder
  $\rightarrow$ Scalar processors execute the same instruction at a time
- Optimized for computation

Graphics Processing Unit (2)



Example: Fermi Architecture of NVIDIA



Picture taken from [Breß et al., 2013b]

GPU Performance Pitfalls

Data transfers between host and device:

- One of the most important performance factors in GPU programming
  $\rightarrow$ All data has to pass across the PCIexpress bus
  $\rightarrow$ bottleneck
- Limited memory capacity (1 to 16GB)
  $\rightarrow$ Efficient memory management necessary

$\rightarrow$ GPU algorithm <u>can</u> be faster than its CPU counterpart

[Gregg and Hazelwood, 2011]

Summary: CPU vs. GPU

CPU is likely to be better if

- Algorithm needs much control flow or cannot be parallelized
- Data set is relatively small or exceeds capacity of GPU RAM

GPU is likely to be better if

- Algorithm can be parallelized and need moderate control flow
- Data set is relatively large but still fits in the GPU RAM

Rule of Thumb:

- Use CPU for little and GPU for large datasets

Graphics Processing Unit: Programming Model

How to program a GPU? (1)

GPUs are programmed using the *kernel programming model*.

Kernel:

- Is a simplistic program
- Forms the basic unit of parallelism
- Scheduled concurrently on several scalar processors in a SIMD fashion $\rightarrow$ Each kernel invocation (thread) executes the same code on its own share of the input

Workgroup:

- Logically grouping of all threads running on the same multiprocessor

How to program a GPU? (2)

Host Code:

- Executed on the CPU
- Manages all processing on the GPU

Device Code:

- The *kernel*, is the GPU program
- Executed massively parallel on the GPU
- General limitations: no dynamic memory allocation, no recursion

Processing Data on a GPU: Basic Structure

1. CPU instructs to copy all data needed for a computation from the RAM to the GPU RAM
2. CPU launches the GPU kernel
3. CPU instructs to copy the result data back to CPU RAM

Processing Data on a GPU: Basic Structure (2)

- CPU may wait (synchronous kernel launch) or perform other computations (asynchronous kernel launch) while the kernel is running
- GPU executes the kernel in parallel
- GPU can only process data located in its memory
  $\rightarrow$ Manual data placement using special APIs

Frameworks for GPU Programming

Compute Unified Device Architecture (CUDA):

- NVIDIA's Architecture for parallel computations
- Program GPUs in CUDA C using the CUDA Toolkit

Open Computing Language (OpenCL):

- Open Standard
- Targets parallel programming of heterogeneous systems
- Runs on a broad range of hardware (CPUs or GPUs)

Graphics Processing Unit: General Problems for Data Processing

GPU-accelerated DBMS: General Problems

1. Data placement strategy
2. Predicting the benefit of GPU acceleration
3. Force in-memory database
4. Increased complexity of query optimization

[Breß et al., 2013b]

Data placement strategy

Problem:

- Data transfer between CPU and GPU is the main bottleneck
- GPU memory capacity limited → database does not fit in GPU RAM

Data placement:

- GPU-accelerated databases try to keep relational data cached on the device to avoid data transfer
- Only possible for a subset of the data
- **Data placement strategy:** Deciding which part of the data should be offloaded to the GPU
  → Difficult problem that currently remains unsolved

Predicting the benefit of GPU acceleration

- Operators may generate a large result
- Often unfit for GPU-offloading
- Result size of an operation is typically not known before execution (estimation errors propagate through the query plan, estimation is typically bad for operations near the root)

→ Predicting whether a given operator will benefit from the GPU is a hard problem

Force in-memory database

- GPU-accelerated operators are of little use, when most time is spent on disk I/O
- Time savings will be small compared to the total query runtime
- GPU improves performance only once the data is in main memory
- Disk-resident databases are typically very large, making it harder to find an optimal data placement strategy

Increased complexity of query optimization

Option of running operations on a GPU increases the complexity of query optimization:

- The plan search space is drastically larger
- Require cost function that compares run-times across architectures
- GPU-aware query optimization remains an open challenge

Graphics Processing Unit: Architectural Considerations for DBMS

Row Stores vs. Column Stores

Store a Table row wise:

Produkt	Ort	Umsatz	Jahr
Merlot	Magdeburg	4325	2010
Guinness	Magdeburg	2341	2010
Merlot	Ilmenau	5543	2010
Pinot Noir	Ilmenau	4944	2010

Store a Table column wise:

Produkt
Merlot
Guinness
Merlot
Pinot Noir

Ort
Magdeburg
Magdeburg
Ilmenau
Ilmenau

Umsatz
4325
2341
5543
4944

Jahr
2010
2010
2010
2010

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 transfered to GPU RAM

Processing Model

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

- Tuple-at-a-time volcano model [Graefe, 1990]
  - Operator requests next tuple, processes it, and passes it to the next operator
- Operator-at-a-time bulk processing [Manegold et al., 2009]
  - 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
  $\rightarrow$ Making effective usage of the memory hierarchy
  [Manegold et al., 2009]
- Parallelism inside an operator, multiple cores used for processing a single operation
  $\rightarrow$ Intra-operator parallelism

Disadvantages:

- Increased memory requirement, since intermediate results are materialized [Manegold et al., 2009]
- 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 transfered over the PCIe bus by using large memory chunks
- Tuple-wise processing is not possible on the GPU, because inter-kernel communication is undefined [NVIDIA, 2012] $\rightarrow$ No pipelining possible
- Operator-at-a-time processing can be easily combined with operator-wise scheduling

GPU-accelerated Database Operators

State of the Art: GPUs in Databases

GPUs are utilized for accelerating query processing like:

- Relational operations

  [Bakkum and Skadron, 2010, Diamos et al., 2012, Govindaraju et al., 2006, He et al., 2009, He et al., 2008, He and Yu, 2011, Kaldewey et al., 2012, Pirk, 2012, Pirk et al., 2011, Pirk et al., 2012]
- XML path filtering [Moussalli et al., 2011]
- Online aggregation [Lauer et al., 2010]
- Compression [Andrzejewski and Wrembel, 2010, Fang et al., 2010]
- Scans [Beier et al., 2012]

GPUs are as well utilized for accelerating query optimization:
e.g., GPU based selectivity estimation

[Augustyn and Zederowski, 2012, Heimel and Markl, 2012]

Co-processoring in a DBMS

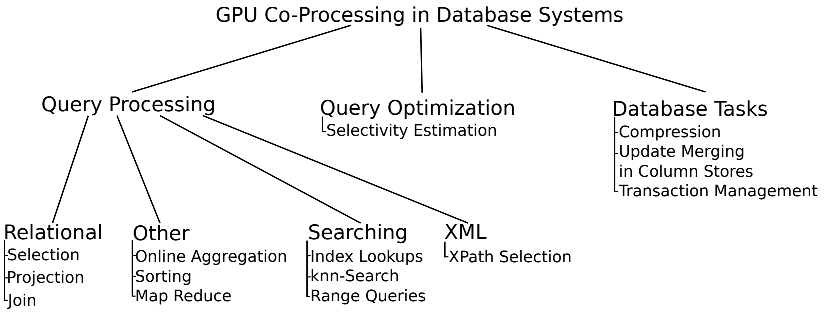


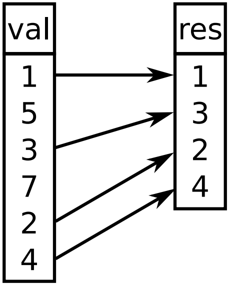
Figure: Classification of Co-Processing Approaches

[Breß et al., 2013a]

Selection

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

pred: val<5



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

Selection

How to parallelize a selection efficiently?

Points to consider:

- Concurrent writes may corrupt data structures
  $\rightarrow$ Usually, correctness is ensured by locks
- Locks may serialize your threads and nullify the performance gained by parallel execution
  $\rightarrow$ We need a way to ensure correctness and consistency without using locks

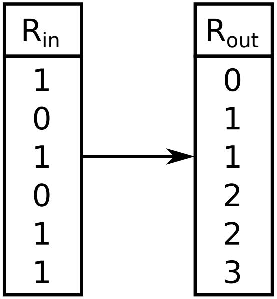
$\rightarrow$ pre-compute write locations

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

[He et al., 2009]

Example: Prefix Sum



Parallel Selection

- 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

[He et al., 2009]

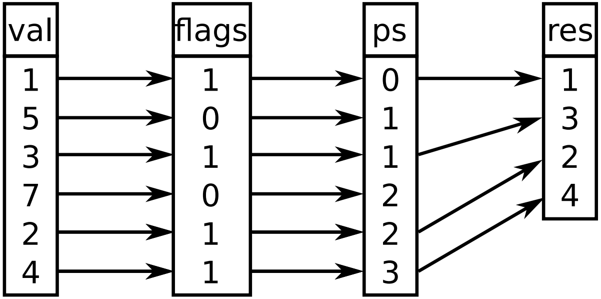
Parallel Selection

- 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];
  }
}
```

[He et al., 2009]

Parallel Selection: Example



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

compute prefix sum from *flags*

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

Joins

- Non Indexed Nested Loop Join
- Indexed Nested Loop Join
- Sort Merge Join
- Hash Join

We will not discuss the individual parallelism strategies, but we focus on the common problems here!

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

Joins

Joins use a three-step output scheme:

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 the their write locations

$\rightarrow$ lock free processing scheme

[He et al., 2009]

Outlook

Hybrid Query Processing Engine

Hybrid Query Processing Engine (HyPE):

- Distributes database operations response time minimal on CPU/GPU
- Tries to utilize the processing device most suited for an operation while keeping track of the load situation

[Breß et al., 2012c, Breß et al., 2012a, Breß et al., 2012d, Breß et al., 2012b, Breß et al., 2013a]

CoGaDB

CogaDB (Column Oriented GPU accelerated DBMS):

- Designed as In-Memory Column Store
- Basis for investigating different query optimization strategies [Breß et al., 2012d]
- Prototype for advanced co-processing techniques for column-oriented DBMS [Breß et al., 2013b]

Open Research Questions:

- How can GPU-acceleration be integrated in column stores, and – in particular – how should an efficient data-placement and query optimization strategy for a GPU-aware DBMS look like?

- Which parts of a database engine should be hardware-conscious (fine-tuned to a particular architecture), and which parts should be hardware-oblivious (implemented in a general framework like OpenCL, that can be mapped to multiple architectures at runtime)?

[Breß et al., 2013b]

Open Research Questions:

- How does the performance differ when comparing distributed-query-processing approaches with tailor-made approaches for hybrid CPU/GPU systems?
- What is a suitable transaction protocol that ensures ACID properties over all (co-)processors?
- Is it feasible to include GPU-acceleration in an existing DBMS by changing the architecture successively (e.g., Ocelot) or are the necessary changes on DBMS architecture and software so invasive and expensive that a rewrite from scratch is necessary (e.g., CoGaDB)?

[Breß et al., 2013b]

Invitation

Your are invited to join our research on GPU-accelerated data management, e.g., in form of:

- Bachelor or master thesis
- Student assistant (Hilfswissenschaftler)

Contact: Sebastian Breß
(`sebastian.bress@cs.tu-dortmund.de`)

Thank you for your attention!

Are there any questions or suggestions?

References I

Andrzejewski, W. and Wrembel, R. (2010).
GPU-WAH: Applying GPUs to Compressing Bitmap Indexes with Word Aligned Hybrid.
In *DEXA (2)*, pages 315–329.

Appuswamy, R., Gkantsidis, C., Narayanan, D., Hodson, O., and Rowstron, A. (2013).
Nobody ever got fired for buying a cluster.
Technical Report MSR-TR-2013-2, Microsoft Research, Cambridge, UK.

Augustyn, D. R. and Zederowski, S. (2012).
Applying CUDA Technology in DCT-Based Method of Query Selectivity Estimation.
In *GID*, pages 3–12. Springer.

Bakkum, P. and Skadron, K. (2010).
Accelerating SQL database operations on a GPU with CUDA.
In *GPGPU*, pages 94–103. ACM.

Beier, F., Kilias, T., and Sattler, K.-U. (2012).
GiST Scan Acceleration using Coprocessors.
In *DaMoN*, pages 63–69. ACM.

Breß, S., Beier, F., Rauhe, H., Sattler, K.-U., Schallehn, E., and Saake, G. (2013a).
Efficient co-processor utilization in database query processing.
Information Systems, 38(8):1084–1096.
http://dx.doi.org/10.1016/j.is.2013.05.004.

References II

Breß, S., Beier, F., Rauhe, H., Schallehn, E., Sattler, K.-U., and Saake, G. (2012a).
Automatic Selection of Processing Units for Coprocessing in Databases.
In *ADBIS*, pages 57–70. Springer.

Breß, S., Geist, I., Schallehn, E., Mory, M., and Saake, G. (2012b).
A framework for cost based optimization of hybrid cpu/gpu query plans in database systems.
Control and Cybernetics, 41(4):715–742.

Breß, S., Heimel, M., Siegmund, N., Bellatreche, L., and Saake, G. (2013b).
Exploring the design space of a GPU-aware database architecture.
In *ADBIS workshop on GPUs In Databases (GID)*, pages 225–234. Springer.

Breß, S., Mohammad, S., and Schallehn, E. (2012c).
Self-Tuning Distribution of DB-Operations on Hybrid CPU/GPU Platforms.
In *GvD*, pages 89–94. CEUR-WS.

Breß, S., Schallehn, E., and Geist, I. (2012d).
Towards Optimization of Hybrid CPU/GPU Query Plans in Database Systems.
In *GID*, pages 27–35. Springer.

Diamos, G., Wu, H., Lele, A., Wang, J., and Yalamanchili, S. (2012).
Efficient Relational Algebra Algorithms and Data Structures for GPU.
Technical report, Center for Experimental Research in Computer Systems (CERS).

technische universität
dortmund

References III

Fang, W., He, B., and Luo, Q. (2010).
Database Compression on Graphics Processors.
PVLDB, 3:670–680.

Govindaraju, N., Gray, J., Kumar, R., and Manocha, D. (2006).
GPUTeraSort: High Performance Graphics Coprocessor Sorting for Large Database Management.
In *SIGMOD*, pages 325–336. ACM.

Graefe, G. (1990).
Encapsulation of parallelism in the volcano query processing system.
In *SIGMOD*, pages 102–111. ACM.

Gregg, C. and Hazelwood, K. (2011).
Where is the data? Why You Cannot Debate CPU vs. GPU Performance without the Answer.
In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software*, ISPASS '11, pages 134–144. IEEE.

He, B., Lu, M., Yang, K., Fang, R., Govindaraju, N. K., Luo, Q., and Sander, P. V. (2009).
Relational Query Coprocessing on Graphics Processors.
ACM Trans. Database Syst., 34:21:1–21:39.

He, B., Yang, K., Fang, R., Lu, M., Govindaraju, N., Luo, Q., and Sander, P. (2008).
Relational Joins on Graphics Processors.
In *SIGMOD*, pages 511–524. ACM.

References IV

He, B. and Yu, J. X. (2011).
High-Throughput Transaction Executions on Graphics Processors.
PVLDB, 4(5):314–325.

Heimel, M. and Markl, V. (2012).
A First Step Towards GPU-assisted Query Optimization.
In *ADMS*.

Kaldewey, T., Lohman, G., Mueller, R., and Volk, P. (2012).
GPU Join Processing Revisited.
In *DaMoN*, pages 55–62. ACM.

Lauer, T., Datta, A., Khadikov, Z., and Anselm, C. (2010).
Exploring Graphics Processing Units as Parallel Coprocessors for Online Aggregation.
In *DOLAP*, pages 77–84. ACM.

Manegold, S., Kersten, M. L., and Boncz, P. (2009).
Database architecture evolution: Mammals flourished long before dinosaurs became extinct.
PVLDB, 2(2):1648–1653.

Moussalli, R., Halstead, R., Salloum, M., Najjar, W., and Tsotras, V. J. (2011).
Efficient XML Path Filtering Using GPUs.
In *ADMS*.

References V

NVIDIA (2012).
NVIDIA CUDA C Programming Guide.
http://developer.download.nvidia.com/compute/DevZone/docs/html/C/doc/CUDA_C_Programming_Guide.pdf.
pp. 30–34, Version 4.0, [Online; accessed 1-May-2012].

Pirk, H. (2012).
Efficient Cross-Device Query Processing.
Proceedings of the VLDB Endowment.

Pirk, H., Manegold, S., and Kersten, M. (2011).
Accelerating Foreign-Key Joins using Asymmetric Memory Channels.
In *ADMS*, pages 585–597. VLDB Endowment.

Pirk, H., Sellam, T., Manegold, S., and Kersten, M. (2012).
X-Device Query Processing by Bitwise Distribution.
In *DaMoN*, pages 48–54. ACM.