Challenges for a GPU-Accelerated Dynamic Programming Approach for Join-Order Optimization

Andreas Meister, Gunter Saake
Otto-von-Guericke-University Magdeburg
Institute for Technical and Business Information Systems
Magdeburg, Germany
firstname.lastname@ovgu.de

ABSTRACT

Relational database management systems apply query optimization in order to determine efficient execution plans for declarative queries. Since the execution time of equivalent query execution plans can differ by several orders of magnitude based on the used join order, join-order optimization is one of the most important problems within query processing. Since the time-budget of query optimization is limited, efficient join-order optimization approaches are needed to determine execution plans with low execution times. The state of the art in commercial systems for determining optimal join orders is dynamic programming. Unfortunately, existing algorithms are mainly sequential algorithms, which do not benefit from current parallel system architectures. In current system architectures, specialized co-processors, such as GPUs, provide a higher computational power compared to CPUs. If the full potential of GPUs is used, query optimizer can provide optimal solutions for more complex problems. Unfortunately, adapting existing dynamic programming approaches for join-order optimization to GPUs is not straightforward. In this paper, we discuss the challenges for a GPU-accelerated dynamic programming approach for join-order optimization, and propose different ways to handle these challenges.

Categories and Subject Descriptors

H.2.4 [**Database Management**]: Systems—*Relational Databases*; D.1.6 [**Numerical Analysis**]: Optimization—*Global Optimization*

General Terms

GPGPU, Database Optimization

Keywords

GPU-Accelerated Optimization, Join-Order Optimization, Dynamic Programming

28th GI-Workshop on Foundations of Databases (Grundlagen von Datenbanken), 24.05.2016 - 27.05.2016, Noerten-Hardenberg, Germany.

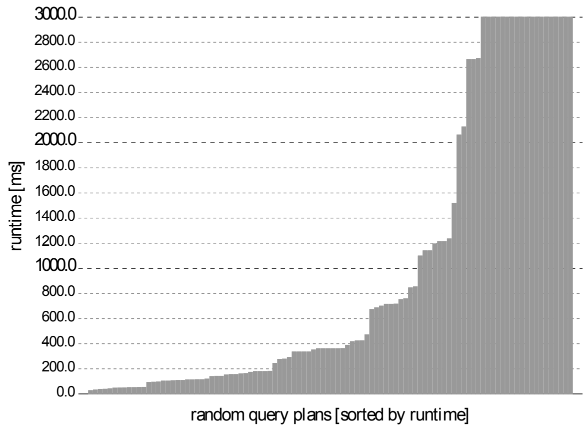

Figure 1: Execution times of different join orders for Query 5 of TPC-H benchmark adapted from [17]. Execution of queries can vary by several orders of magnitude depending on the join order.

1. INTRODUCTION

Relational Database Management Systems (DBMSs) use declarative query languages, such as the Structured Query Language (SQL). Within declarative query languages, queries only specify what data should be retrieved, but not how the data should be retrieved by the system. Therefore, the system needs to transform the declarative query into an executable plan. Based on the properties of the relational operators, such as commutativity of joins, several equivalent plans exist. In order to ensure an efficient query processing, DBMSs need to select an efficient query execution plan. To this end, DBMSs apply different heuristics, such as pushing down selections, and optimization approaches to ensure the selection of efficient query execution plans. The execution time of a query can vary by several orders of magnitude based on the join order [16], like in Query 5 of the TPC-H benchmark [17], see Figure 1. Therefore, one of the most important optimization problems within the query processing is join-order optimization.

Because join-order optimization is an NP-complete problem [22], providing optimal solutions is challenging. Often heuristics, such as avoiding Cartesian products or evaluating only left-deep trees, are used in order to cope with the complexity of join-order optimization by reducing the search space [26]. Unfortunately, by reducing the search space also efficient execution plans or even the optimal execution plan

Year	CPU	GFLOPS	AMD GPU	GFLOPS
2002	Pentium 4 (Northwood)	12.2	9700 Pro	31.2
2003	Pentium 4 (Northwood)	12.8	9800 XT	36.5
2004	Pentium 4 (Prescott)	15.2	X850 XT	103.7
2005		15.2	X1800 XT	134.4
2006	Core 2 Duo	23.4	X1950	375.0
2007	Core 2 Quad	48.0	HD 2900 XT	473.6
2008	Q9650	96.0	HD 4870	1200.0
2009	Core i7 960	102.4	HD 5870	2720.0
2010	Core i7 970	153.6	HD 6970	2703.0
2011	Core i7 3960X	316.8	HD7970	3789.0
2012	Core i7 3970X	336.0	HD 7970 GHz Edition	4301.0

Table 1: Theoretical computational power of CPUs and AMD GPUs adapted from [20]. GPUs provide a higher computational power compared to CPUs

may not be considered during the optimization, leading to inefficiencies within the query execution [26]. In order to provide an optimal join order the dynamic programming approach was proposed [23], which is currently the state of the art approach for providing an optimal join order in commercial systems, such as Oracle[1] or Postgres[2].

As the dynamic programming approach was proposed almost 40 years ago, where only traditional system architectures with single-core CPUs were available, the traditional dynamic programming approach is a sequential algorithm. In the past, this was not a problem, because the clock-speed of CPUs increased every year, and, hence, also the performance and applicability of the dynamic programming approach increased automatically. Unfortunately, based on physical effects, such as the power wall [2], increasing the clock speed is not practicable anymore. In order to provide more computational power, CPU vendors started to combine multiple cores on one chip. With this approach the computational capacity of CPUs was increased in the past years, see Table 1. Unfortunately, sequential algorithms do not benefit from parallel processors, and, hence, the applicability of the dynamic programming approach for join-order optimization is practically limited to the optimization of queries with up to 12 tables [7]. To utilize the potential powers of multi-core CPUs, the execution of existing sequential algorithms need to be parallelized. Han et al. proposed a parallel algorithm for multi-Core CPUs in order to apply the dynamic programming approach to more complex optimization problems [7]. By using different partitioning schemata for assigning calculations to available CPU cores, Han et al. achieve almost linear speedup and extend the practicable limit of the dynamic programming approach to up to 20-25 tables depending on the topology of the queries [7]. Unfortunately, increasing the numbers of CPU cores on one chip is also limited. Based on the fixed energy-budget of CPUs, the maximal number of CPU cores on one chip is estimated to be between ten to around one hundred cores per chip [8].

[1]oracle.de

[2]postgresql.org/

To provide further computational power, the use of co-processors was proposed. Co-processors, such as GPUs, FPGAs, and MICs, are specialized for specific calculations [4]. For example, GPUs are highly parallel co-processors, offering a higher computational power per dollar compared to CPUs based on their specialized architecture [18], see Table 1. In order to use the computational power of GPUs, we need to adapt existing algorithms to the specialized architecture of GPUs, similar to the change from single-core CPUs to multi-core CPUs.

Many GPU-adapted approaches in the field of optimization [5, 19] and DBMSs [3, 9, 11], show that the adaption is worthwhile and high speedups are possible. Unfortunately, in the field of query optimization only approaches for selectivity estimation [10] are available. Other important optimization problems, such as join-order optimization are still lacking GPU-based approaches. We argue that GPU-acceleration will be benefiting also for other optimization approaches within the field of query optimization in DBMSs [15] and, hence, plan to adapt the join-order optimization on GPUs, starting with the dynamic programming approach [14].

Unfortunately, the adaption of approaches from CPU-based to GPU-based execution is challenging based on the different architectures and resulting properties of GPUs and CPUs. Therefore many challenges arise during the adaption of the dynamic programming approach for join-order optimization. In this paper, we discuss the challenges for a GPU-accelerated dynamic programming approach for join-order optimization, and propose different ways to handle these challenges.

The remainder of this paper is structured as follow. In Section 2, we discuss the difference between CPUs and GPUs. In Section 3, we explain the basic concept of dynamic programming approach. In Section 4, we illustrate the challenges for a GPU-based dynamic programming approach for join-order optimization. In the last section, we conclude our discussions.

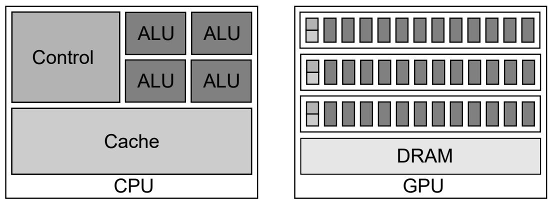

Figure 2: Simplified architectures of CPUs and GPUs adapted from [6]. The architecture of CPUs and GPUs highly differ.

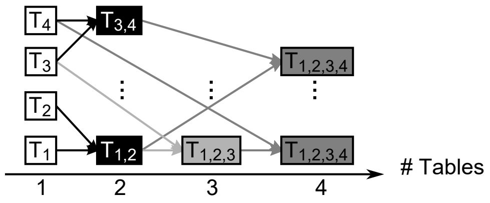

Figure 3: Execution schema of the dynamic programming approach for join-order optimization. Solutions for complex problems are provided by combining existing solutions.

2. GPU-ACCELERATION

The architecture for CPUs and GPUs highly differ, see Figure 2, based on different execution focus.

CPUs efficiently support a wide range of applications from control-intensive workflows to simple mathematical operations. Therefore, CPUs provide advanced techniques to support the broad variance of applications. CPUs provide branch-prediction techniques to support control intensive applications, and pipelining in order to increase the throughput, when multiple operations are executed concurrently. These techniques require additional functionality. Hence, a large part of CPU chips are used for the control logic to provide this required functionality. In order to provide a low response time for multiple concurrent operations, CPU cores are independent, whereby each core has a high clock rate up to four GHz. Based on the high clock-speed and independent execution, only few cores (currently up to 18 cores) can be put on one CPU chip. A CPU chip can directly access the main memory. Since the access to the main memory is slow compared to the access of the internal register and clock speed of CPUs, large caches (up to 45 MB) are used in order to reduce the access gap to main memory. Hereby, each core has access to the complete shared cache.

In contrast to CPUs, GPU cores cannot directly access the main memory. Therefore, GPUs provide their own memory on the device, accessible by the GPU cores. Before the GPU cores can process data, the data needs to be transferred from the main memory to the device memory of the GPU via the Peripheral Component Interconnect Express (PCIe) bus. Although GPUs provide a high computational power based on the number of cores (up to around 5000), the computational power and also the capabilities of a single GPU core is limited compared to CPU cores. In order to manage the high number of cores, GPU cores cannot work independently. GPU cores are grouped to streaming processors, whereby the number of cores can vary depending on the architecture of the GPU. For each streaming processor, only one control unit and cache is provided. Hence, all cores of one streaming processor need to execute the same instruction. Additionally, each core of a streaming processor has a lower clock speed (up to 900 Mhz) compared to CPU cores, and lacks such advanced techniques such as branch prediction or pipelining. Based on the simple core model, switches between different executions can be performed without much overhead. The execution of operations and memory transfer from and to the GPU device for general purpose computations are managed by the CPU. The management of executions by the CPU are performed by C-like application programming interfaces (APIs), such as CUDA[3] and OpenCL[4], provided by all GPU vendors.

Based on the different architecture, CPUs and GPUs are suitable for different application scenarios. CPUs are suitable for control-intensive applications and executing different operations on few data items. Whereas GPUs are well suited for parallel calculations, where operations are performed on a huge data set in parallel.

As already mentioned, Han et al. proved with their CPU-based dynamic programming approach that the dynamic programming approach for join-order optimization benefits from parallelization [7]. Unfortunately, the CPU-based dynamic programming approach proposed by Han et al. is not directly applicable to GPUs, based on the different architecture of GPUs compared to CPUs. In order to understand the challenges of adapting the dynamic programming approach for GPUs, we will first explain the approach of dynamic programming for join-order optimization.

3. DYNAMIC PROGRAMMING FOR JOIN-ORDER OPTIMIZATION

The dynamic programming approach for join-order optimization [23] is the state of the art approach for providing an optimal solution within commercial DBMSs, such as Oracle or Postgres. In contrast to randomized approaches, such as genetic algorithms [1], the dynamic programming approach is a deterministic approach, providing always the same solution for the same input. The dynamic programming approach applies an exhaustive search in order to determine the optimal solution for the join-order problem. In order to avoid the evaluation of non-optimal join-orders, the dynamic programming approach solves an optimization problem, by splitting up the problem into subproblems. The splitting of a complex problem into subproblems is done based on the assumption that an optimal solution can only contain optimal subsolutions. Hence, the subproblems are solved in an optimal way, and combined to solutions for more complex problems or the optimal solution for the overall optimization problem, see Figure 3. Based on the combination of existing subsolutions, the dynamic programming approach avoids the calculation of non-optimal join orders in contrast to a brute-force approach. Since, in general, for each subsolution multiple equivalent join-orders are available, the optimal subsolution must be selected based on a cost model. Based on the focus of the optimization, different cost models, such as page accesses [25], communication overhead [13],

[3]developer.nvidia.com/cuda-zone
[4]khronos.org/opencl/

execution time [24], or number of intermediate results [12] can be used.

For the join-order optimization this means that the dynamic programming approach starts with determining the cost of accessing each individual table, see Algorithm 1 line 1 to 4. In DBMSs, different options to access tables are available (e.g., table scan or indexes). Hence, all options need to be evaluated and the best option need to be selected via pruning, see Algorithm 1 line 3. Hereby, for example different access operators can be evaluated, such as table access or index access. After determining the access costs of the table the second iteration combines two tables by a join. In the third iteration, one solution of the first and one solution of the second iteration is combined, whereby the validity of the intermediate results must be checked such as that both solutions do not contain identical tables. Within each iteration, we construct solutions containing exactly one table more than the previous iteration, see Algorithm 1 line 5 to 14. Similar to the table access, multiple equivalents solutions are created. By pruning, we select the optimal solution, see Algorithm 1 line 11.

Algorithm 1 Sequential dynamic programming approach for join-order optimization

```
Input: Query Q joining n tables $(t_1, \cdots, t_n)$
Output: Optimal join order for query Q
 1: for $i = 1$ to $n$ do
 2:     $optimal\_results[t_i] += create\_access\_plans(t_i);$
 3:     $prune\_plans(optimal\_results[t_i]);$
 4: end for
 5: for $i = 2$ to $n$ do
 6:     for all $s \subseteq \{t_1, \cdots, t_n\}$ with $|s| = i$ do
 7:         $optimal\_results[s] = \{\};$
 8:         for all $t_k \in s$ do
 9:             $optimal\_results[s] +=$
10:             $create\_join(optimal\_results[s - \{t_k\}], t_k);$
11:             $prune\_plans(optimal\_results[s]);$
12:         end for
13:     end for
14: end for
        return $optimal\_results[t_1, \cdots, t_n]$
```

Since the evaluation of one iteration is dependent on all previous iterations, the dynamic programming approach is only parallelizable within one partition, but not over all partitions [7], see Algorithm 1 line 6 to 13. Han et al. used this limited parallelism to speed up the dynamic approach using multi-core CPUs. The basic idea of their approach is to determine within each iteration, which calculations need to be evaluated. The necessary calculations are partitioned based on the number of available CPU-cores. The cores can execute the calculation independent of each other. When all CPU cores finished their execution, the results of all cores are merged. In the next step, the merged results are pruned, so that only the optimal solution for one subproblem is stored. This continues until the final result can be provided. In order to avoid an unbalanced load between processors, they evaluated several partition schemata. Based on a proposed data structure, invalid solutions, where the tables of the two join partners are overlapping, are skipped. Using the parallel execution of multi-core CPUs and their adapted approach, Han et al. achieve an almost linear speed up for the dynamic programming approach on multi-core CPUs.

Based on the parallel architecture of GPUs, GPUs provide a higher computational power compared to CPUs. Since the dynamic programming approach benefits from the parallel execution on CPUs, we also claim that the dynamic programming approach will benefit from the parallel execution on GPUs.

4. CHALLENGES FOR DP ON GPUS

Although a parallel dynamic programming approach exists for multi-core CPUs, it is not directly applicable to GPUs. In the following, we will explain why the adaption of the dynamic programming approach is challenging.

Although GPUs offer the advantage of an high theoretical computational power, reaching the peak performance of GPUs is a cumbersome and challenging task. Hereby, the main challenge is the efficient use of the architecture, which is completely different to CPUs, cf. Section 2. In order to provide an efficient application on GPUs, applications need to consider the following aspects:

- Avoid branching
- Transfer bottleneck
- Memory hierarchy
- Parallel calculations
- Limited storage

In the following, we will describe these challenges and propose how to handle these challenges for the dynamic programming approach on GPUs.

Avoid branching: As mentioned previously, the GPU cores are grouped to streaming processors. Since cores of one streaming processor share cache and control logic of the streaming processor, all cores of one streaming processor can only execute the same instruction. If cores of a streaming processor need to execute different operations, for example based on branching, the different operations will be executed sequentially. While sequentially executing different operations, one part of cores of the streaming processor stalls, while the other part of the group executes an operation. Since only part of the cores are active, not the complete computational potential of GPUs is used, reducing the efficiency and also the usefulness of GPUs. Hence, branching should be avoided for GPU-accelerated applications. For the dynamic programming approach, this means that we should try to eliminate invalid calculations, and maybe not simply execute and skip invalid calculations as done in the approach for multi-core CPUs of Han et al. [7].

Transfer bottleneck: Similar to CPUs, data need to be available in the caches of streaming processors for the processing. Unfortunately, the data cannot be loaded directly into the cache of streaming processors on GPUs. Before a GPU can load the data into the cache of streaming processors, the data need to be stored in the device memory of GPUs. To store data on the device memory, data need to be transferred from the main memory to the device memory via the PCIe bus. Unfortunately, the bandwidth of the PCIe bus is limited compared to the internal memory bandwidth of GPUs. Therefore, the PCIe bus poses a major bottleneck for data intensive applications. GPU-accelerated applications should avoid the inefficient transfer from main memory to GPU device memory or should overlap the transfer with calculations to hide the limited bandwidth of the PCIe bus. For the dynamic programming approach, this means that we should perform all calculations on the GPU if possible. Hence, only statistics, such as selectivity estimations and cardinality of tables, need to be transferred at the beginning to the GPU, and, at the end, only the optimized plan should be transferred back to the main memory. Since statistics do not need to be transactional consistent, the caching of these statistics on the GPU is also an alternative to further reduce the communication via the PCIe bus.

Memory hierarchy: When the data is available on the device, the data can be processed by the streaming processor of GPUs. In contrast to CPUs, in GPUs, there are different types of memory available, providing different access speed and cacheability. The memory of GPUs consists of global memory, constant memory, texture memory, shared and local memory [21]. Global memory is the largest available storage, whereas the access speed of read and write operations is the lowest. In contrast to global memory, constant memory is a read only memory. Although constant memory also uses the global memory, streaming processors provide special caches for constant memory. Therefore, the access speed to constant memory is increased by a better cacheability. Similar to constant memory, texture memory is read only memory, which is cached for streaming processors. In contrast to constant memory, the texture memory is optimized for two dimensional data. Shared memory resides on streaming processors, hereby, each core of the streaming processor can directly access the shared memory. Since shared memory resides on streaming processors, the size is smaller than the global memory, but the access speed is faster. Local memory is only accessible by one specific core. Unfortunately, there is no dedicated memory available for this type of memory, and, hence, global memory is used. Since constant memory and texture memory provide good cacheability and shared memory is fast accessible, the usage of these memory types is essential for an efficient execution on GPUs. For the dynamic programming approach, this means that we should transfer results of each iteration into the cacheable memory (Constant or texture memory). For the transfer from global memory to the constant memory coalesced memory accesses should be used, because coalesced accesses are more efficient on GPUs compared to sequential memory accesses. In addition to the use of cacheable memory, shared memory should be used during the needed merge and reduction phase.

Parallel calculations: As pointed out, we need to avoid branching, avoid transfers from and to main memory, and use the fast on-chip memory to achieve the peak performance of GPUs by parallel calculations. When we consider a GPU-accelerated dynamic programming approach, providing enough parallel calculations poses a challenge. As mentioned previously, the execution of the dynamic programming approach requires that all previous iterations are finished before evaluating a new iteration. The consequence for a GPU-accelerated dynamic programming approach is that enough calculations need to be available in order to utilize the high computational power of GPUs. Depending on the topology of queries, this is easily fulfilled, see Figure 4. Although for linear and cyclic query topologies, only a small number of intermediate results need to be evaluated for an optimal result (2680 and 7240 for 20 tables), the number of intermediate results, which we need to evaluate, explodes when clique query topology or queries with cross-joins are considered (3.5 billion for 20 tables). For the dynamic programming approach, this means that, on GPUs, we should especially focus on the compute intensive optimization, such as clique or star query topologies or queries with cross joins.

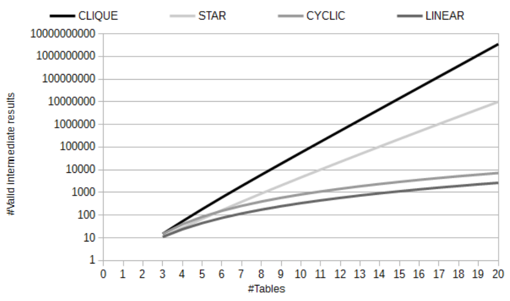

Figure 4: Number of valid intermediate results of the dynamic programming approach. Especially, clique and star queries need to evaluate a high number of intermediate results.

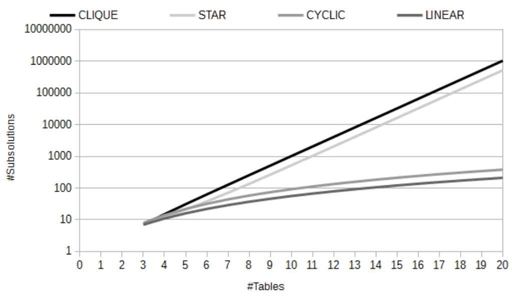

Figure 5: Number of stored intermediate results of the dynamic programming approach. Especially, clique and star queries have high storage requirements.

Limited storage: Unfortunately, with the number of intermediate results also the storage requirements for the dynamic programming approach rises. In our previous work, we argued that the query optimization only requires little storage sizes to provide results of their optimization [14]. Although for the dynamic programming approach, this is true for the inputs, it might not be true for the storage of the optimal subsolutions, see Figure 5. Although the output of the dynamic programming approach is only one optimal plan for a given query, the dynamic programming approach need to store intermediate results in order to ensure an efficient optimization. Hence, for every solution of a subproblem one result need to be stored. Although Vance et al. argue that for one solution only 16 bytes need to be stored [26], this can already be challenging for larger queries. In Figure 5, we show the number of intermediate results, stored during the execution of the dynamic programming approach. Whereas for linear, and cyclic queries the number of solution is small and manageable (211 and 382 for 20 tables), the number of intermediate results explodes, when we consider cross joins, or clique and star query topologies (1 million, 1 million and 0,5 million for

20 tables). Since even high end class GPUs provide only a small device memory (24 GB for the Nvidia Tesla K 80), the storage requirements for the intermediate results poses a further challenge. For the dynamic programming approach, this means that on the one hand, we need an efficient storage structure for storing intermediate results. On the other hand, we need mechanisms to partition the evaluation of iterations for queries with higher numbers of tables similar to the vectorized executions in DBMSs [27].

Although a high number of challenges exists for the adaption of the dynamic programming approach for join-order optimization on GPUs, we can use existing techniques and considerations to solve these challenges. By solving these challenges, we can use the GPU-acceleration for the dynamic programming approach to extend the applicability to more complex optimization problems and to reduce the optimization time for simple optimization problems.

5. SUMMARY

Within this paper, we discuss the challenges for a GPU-accelerated dynamic programming approach for join-order optimization. Based on the properties of GPUs and the execution model of the dynamic programming approach different challenges arise. Although the challenges for a GPU-accelerated application, such as branching, transfer bottleneck, memory hierarchy, parallel calculations and limited storage amount, are well known, within each algorithms these challenges have to be handled in different ways. Hence, we proposed different ways how to handle and avoid these different challenges.

6. REFERENCES

[1] K. Bennett, M. C. Ferris, and Y. E. Ioannidis. A Genetic Algorithm for Database Query Optimization. ICGA, pages 400–407. Morgan Kaufmann Publishers, 1991.

[2] S. Borkar and A. A. Chien. The future of microprocessors. *CACM*, 54(5):67–77, May 2011.

[3] S. Breß, N. Siegmund, M. Heimel, M. Saecker, T. Lauer, L. Bellatreche, and G. Saake. Load-Aware Inter-Co-Processor Parallelism in Database Query Processing. *Data & Knowledge Engineering*, 2014.

[4] D. Broneske, S. Breß, M. Heimel, and G. Saake. Toward Hardware-Sensitive Database Operations. In *EDBT*, pages 229–234. OpenProceedings.org, 2014.

[5] J. M. Cecilia, J. M. García, A. Nisbet, M. Amos, and M. Ujaldón. Enhancing data parallelism for Ant Colony Optimization on GPUs. *Journal of Parallel and Distributed Computing*, 73(1):42–51, 2013.

[6] G. K. Chen and Y. Guo. Discovering epistasis in large scale genetic association studies by exploiting graphics cards. *Frontiers in Genetics*, 4(266), 2013.

[7] W.-S. Han, W. Kwak, J. Lee, G. M. Lohman, and V. Markl. Parallelizing Query Optimization. *PVLDB*, 1(1):188–200, 2008.

[8] N. Hardavellas, M. Ferdman, B. Falsafi, and A. Ailamaki. Toward Dark Silicon in Servers. *IEEE Micro*, 31(4):6–15, July 2011.

[9] B. He, M. Lu, K. Yang, R. Fang, N. K. Govindaraju, Q. Luo, and P. V. Sander. Relational Query Coprocessing on Graphics Processors. *TODS*, 34:21:1–21:39, 2009.

[10] M. Heimel, M. Kiefer, and V. Markl. Self-Tuning, GPU-Accelerated Kernel Density Models for Multidimensional Selectivity Estimation. SIGMOD, pages 1477–1492. ACM, 2015.

[11] T. Karnagel, R. Müller, and G. M. Lohman. Optimizing GPU-accelerated Group-By and Aggregation. In *ADMS*, pages 13–24. ADMS, 2015.

[12] V. Leis, A. Gubichev, A. Mirchev, P. Boncz, A. Kemper, and T. Neumann. How Good Are Query Optimizers, Really? *PVLDB*, 9(3):204–215, Nov. 2015.

[13] L. F. Mackert and G. M. Lohman. R* Optimizer Validation and Performance Evaluation for Distributed Queries. VLDB, pages 149–159. Morgan Kaufmann, 1986.

[14] A. Meister. GPU-accelerated join-order optimization. *VLDB PhD workshop*, 2015.

[15] A. Meister, S. Breß, and G. Saake. Toward GPU-accelerated Database Optimization. *Datenbank-Spektrum*, 15(2):131–140, 2015.

[16] G. Moerkotte, P. Fender, and M. Eich. On the Correct and Complete Enumeration of the Core Search Space. SIGMOD, pages 493–504. ACM, 2013.

[17] T. Neumann. Engineering High-Performance Database Engines. *PVLDB*, 7(13):1734–1741, 2014.

[18] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krüger, A. Lefohn, and T. J. Purcell. A Survey of General-Purpose Computation on Graphics Hardware. *Computer Graphics Forum*, 26(1):80–113, 2007.

[19] P. Pospichal, J. Schwarz, and J. Jaros. Parallel Genetic Algorithm Solving 0/1 Knapsack Problem Running on the GPU. MENDEL, pages 64–70. Brno University of Technology, 2010.

[20] P. Rogers, J. Macri, and S. Marinkovic. AMD heterogeneous Uniform Memory Access. 2013.

[21] S. Ryoo, C. I. Rodrigues, S. S. Baghsorkhi, S. S. Stone, D. B. Kirk, and W.-m. W. Hwu. Optimization Principles and Application Performance Evaluation of a Multithreaded GPU Using CUDA. PPoPP, pages 73–82. ACM, 2008.

[22] W. Scheufele and G. Moerkotte. On the Complexity of Generating Optimal Plans with Cross Products (Extended Abstract). PODS, pages 238–248. ACM, 1997.

[23] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. Access Path Selection in a Relational Database Management System. SIGMOD, pages 23–34. ACM, 1979.

[24] E. J. Shekita, H. C. Young, and K.-L. Tan. Multi-Join Optimization for Symmetric Multiprocessors. VLDB, pages 479–492. Morgan Kaufmann Publishers, 1993.

[25] M. Steinbrunn, G. Moerkotte, and A. Kemper. Heuristic and Randomized Optimization for the Join Ordering Problem. *VLDB Journal*, 6(3):191–208, Aug. 1997.

[26] B. Vance and D. Maier. Rapid Bushy Join-order Optimization with Cartesian Products. SIGMOD, pages 35–46. ACM, 1996.

[27] M. Zukowski, M. van de Wiel, and P. Boncz. Vectorwise: A Vectorized Analytical DBMS. ICDE, pages 1349–1350. IEEE, 2012.