Multi-Level Parallel Query Execution Framework for CPU and GPU

Hannes Rauhe[1,3], Jonathan Dees[2,3], Kai-Uwe Sattler[1], and Franz Faerber[3]

[1] Ilmenau University of Technology
[2] Karlsruhe Institute of Technology
[3] SAP AG

Abstract. Recent developments have shown that classic database query execution techniques, such as the iterator model, are no longer optimal to leverage the features of modern hardware architectures. This is especially true for massive parallel architectures, such as many-core processors and GPUs. Here, the processing of single tuples in one step is not enough work to utilize the hardware resources and the cache efficiently and to justify the overhead introduced by iterators. To overcome these problems, we use just-in-time compilation to execute whole OLAP queries on the GPU minimizing the overhead for transfer and synchronization. We describe several patterns, which can be used to build efficient execution plans and achieve the necessary parallelism. Furthermore, we show that we can use similar processing models (and even the same source code) on GPUs and modern CPU architectures, but point out also some differences and limitations for query execution on GPUs. Results from our experimental evaluation using a TPC-H subset show that using these patterns we can achieve a speed-up of up to factor 5 compared to a CPU implementation.

1 Introduction

The recent development in modern hardware architectures provides great opportunities for database query processing, but at the same time pose also significant challenge to be able to exploit these hardware features. For example, the classic iterator model for query execution (sometimes also called Open-Next-Close-model) together with a tuple-at-a-time processing strategy is no longer optimal for modern hardware, because it does not utilize the cache architecture and does not produce enough work to pay off the overhead in case of parallel processing.

Some of the most prominent techniques for addressing these challenges are fine-granular parallelization and vectorized processing. The goal of parallelization is to speed up query response times and/or improving throughput by exploiting multiple processors/cores. Though, parallel query processing dates back to the mid eighties, modern multi- and many-core processors including GPUs require a rethinking of query execution techniques: processing models range from SIMD to multi-processor strategies, the overhead of starting and coordinating threads is much smaller than on old multi-processors systems and even the cache and memory architectures have changed.

Vectorized processing aims at avoiding the "memory wall", i.e., the imbalance between memory latency and CPU clock speed [3], by using cache-conscious algorithms and data structures, vectorized query processing strategies as used for instance in Vectorwise [2,17] or—as an extreme case—columnwise processing of in-memory database systems like SAP HANA DB.

A promising approach to combine these techniques for making efficient use of the new hardware features is Just-In-Time (JIT) compilation where OLAP queries are translated into a single function that pipelines the data through the operators specified by the query. While lightweight query compilation has been already proposed in other works [12, 14], in our work we particularly focus on compilation for parallelization. This approach requires a different handling of parallelism, because we cannot differentiate between inter- and intra-operator parallelism any more. Instead the parallelism must be an integral part of the code generation.

In this paper, we show that we can use the concept of JIT compilation to build a bulk-synchronous model for parallel query processing that works both on multi-core CPUs and on general purpose GPUs. Although the architectures look very different at first, the techniques we have to use to leverage their full potential are very similar.

Compiling queries into executable code should be driven by some target patterns. As we cannot longer rely on the iterator model, we present and discuss some patterns leveraging the different levels of parallelization (threads at the first level and SIMD at the second level). We extend our previous work on query compilation for multi-core CPUs [6] by taking massively parallel GPU architectures into account and discuss as well as show by experimental results similarities and differences of both processing models and their impact on query execution.

2 Related Work

Over the last decade there has been significant research on using GPUs for query execution. He et al. built GDB, which is able to offload the execution of single operators to the GPU [8]. They implemented a set of primitives to execute joins efficiently. The main problem is that the data has to be transferred to the GPU, but only a small amount of work is done before copying the results back to the main memory. Peter Bakkum and Kevin Skadron ported SQLite to the GPU [1]. The complete database is stored on the graphics card and only the results are transferred. Since all operations are executed on the GPU, their system is not able to use variable length strings. It can process simple filter operations, but does neither support joins nor aggregations. A similar approach with the focus on transactional processing has been tried in [9]. Their implementation is able to execute a set of stored procedures on the data in the GPU's memory and handles read and write access. To the best of our knowledge there are no publications about running complex analytic queries completely on the GPU.

Compiling a complete query plan (or parts of it) into executable code is a radical and novel approach. Classical database systems generate executable code fragments for single operators, which can be used and combined to perform a SQL query. There are several recent approaches for generating machine code for a complete SQL query. The prototype system HIQUE [12] generates C code that is just-in-time-compiled with gcc, linked as shared library and then executed. The compile times are noticeable (around seconds) and the measured execution time compared to traditional execution varies: TPC-H queries 1 and 3 are faster, query 10 is slower. Note that the data layout is not optimized for the generated C code. HyPer [11, 14] also generates code for incoming SQL queries. However, instead of using C, they directly generate LLVM IR code (similar to Java byte code), which is further translated to executable code by an LLVM compiler [13]. Using this light-weight compiler they achieve low compile times in the range of several milliseconds. Execution times for the measured TPC-H queries are also fast, still, they use only a single thread for execution and no parallelization.

One of the fundamental models for parallel query execution is the exchange operator proposed by Graefe [7]. This operator model allows both intra-operator parallelism on partitioned datasets as well as horizontal and vertical inter-operator parallelism. Although this model can also be applied to vectorized processing strategies it still relies on iterator-based execution. Thus, the required synchronization between each operator can induce a noticeable overhead.

3 Basics

In this section we explain the technology as well as some basic decisions we had to make before going into the details of our work. We start with explaining the hardware characteristics in Sect. 3.1. Since we have to copy the data to the GPU's memory before we can use it, the transfer over the PCI-Express bus is the main bottleneck for data intensive algorithms. In Sect. 3.2 we address this problem and explain how we cope with it.

3.1 Parallel CPU and GPU Architecture

Although modern GPU and CPU architectures look different, we face a lot of similar problems when trying to develop efficient algorithms for both. In order to achieve maximum performance, it is mandatory to use parallel algorithms. Also, we generally want to avoid synchronisation and communication overhead by partitioning the input first and then processing the chunks with independent threads as long as possible. In the literature this is called bulk-synchronous processing [16].

There are some differences in the details of parallelism of both platforms. On the CPU we have to be careful not to oversubscribe the cores with a high amount of threads, while the GPU resp. the programming framework for the GPU handles this on its own. However, on the CPU oversubscription can be easily avoided by using tasks instead of threads: a framework, such as Intel's

TBB[1], creates a fixed number of threads once and assigns work to them in form of tasks. This way the expensive context switches between threads are not necessary. While the usage of tasks is enough to exploit the parallelism of CPUs, another form of parallelism is needed to use the GPU efficiently: *SIMD*. Both the OpenCL platform model[2] and CUDA[3] force the developer to write code in a SIMD fashion.

In addition to the parallel design of algorithms, we face the challenge to organize memory access in a way that fits to the hardware. In both architectures, CPU and GPU, there is a memory hierarchy with large and comparatively slow RAM and small but faster caches. Cache-efficient algorithms work in a way, that the data is in the cache when the algorithm needs it and only data that is really needed is transferred to the cache. On the CPU the hardware decides which data is copied from RAM to cache based on fixed strategies. *Local memory* on the GPU is equivalent to cache on the CPU, but it has to be filled manually.

With all data residing in main memory and many cores available for parallel execution, the main memory bandwidth becomes a new bottleneck for query processing. Therefore, we need to use memory bandwidth as efficiently as possible. On modern server CPUs this requires a careful placement of data structures to support NUMA-aware access, which has been shown in [6]. The GPU's memory is highly optimized for the way graphical algorithms access data: threads of neighboring cores process data in neighboring memory segments. If general purpose algorithms work in a similar way, they can benefit from the GPU architecture. If they load data from memory in a different way, the performance can be orders of magnitude worse. Details can be found in [15].

The challenges we face when writing efficient algorithms are very similar, no matter if they run on the GPU or the CPU. Only in case of the GPU the performance effects of using inadequate patterns and strategies are much worse. So we have to make sure, that our algorithms are able to distribute their work over a high number of threads/tasks and exploit the memory hierarchy on both architectures. On the GPU we have to take special care about using a SIMD approach to distribute small work packages and use coalesced access to the device memory. On the CPU it is important not to oversubscribe it with threads and consider NUMA effects of the parallel architecture. In this paper we suggest a model that supports query execution on both architectures and takes care of their differences.

3.2 The Data Transfer Problem

The transfer to the GPU is the main bottleneck for data centric workloads. In general, it takes significantly longer than the execution of the queries, because the transfer of a tuple is usually slower than processing it once. To make matters worse, in general we cannot predict beforehand which values of a column

[1] http://www.threadingbuildingblocks.org
[2] http://www.khronos.org/opencl/
[3] http://www.nvidia.com/cuda

are really accessed and therefore need to transfer all values of a column used by the query, which further increases the data volume. One way to cope with this problem is the usage of Universal Virtual Addressing (UVA). UVA is a sophisticated streaming technique for CUDA, which allows the transfer of required data at the moment it is needed be the GPU. The authors of [10] showed how to use it to process joins on the GPU. However, streaming is still rather inefficient for random access to the main memory. Even if the GPU accesses the main memory sequentially we remain limited to the PCI-Express bandwidth of theoretically 8 GB/s (PCI-Express 2).

From our perspective there are three possibilities to cope with this problem:

1. The data is replicated to the GPU memory before query execution and cached there.
2. The data is stored exclusively in the GPU's memory.
3. CPU and GPU share the same memory.

The first option limits the amount of data to the size of the device memory, which is very small at the moment: high end graphic cards for general-purpose computations have about 4–16 GB. Furthermore, updates have to be propagated to the GPU.

Storing the data exclusively in the GPU's memory also limits the size of the database. Furthermore, there are queries that should be executed on the CPU, because they require advanced string operations or have a large result set (see Sect. 4.4). In this case we face the transfer problem again, only this time we have to copy from GPU to CPU.

The third solution, i.e., CPU and GPU sharing the same memory, is not yet available. Integrated GPUs already use the CPU's main memory, but internally it is divided into partitions for GPU and CPU and each can only access its own partition. So we can copy data very fast from GPU to CPU and vice versa as we remain in main memory, but still need to replicate. For the future AMD, for instance, plans to use the same memory for both parts of the chip [4].

We decided to use the first approach for our research, because it makes no sense to compare GPU and CPU performance as long as we need to transfer the data. We also want to be able to execute queries on the CPU and the GPU at the same time. Until real shared memory is available, replication is the best choice for our needs.

4 A new Model to Execute Queries in Parallel

In this paper we present a framework that provides simple parallelism for just-in-time query compilation. Section 4.1 gives a short overview about why and how we can use JIT compilation to speed up query execution. There are two ways of applying parallelism to the execution. On the one hand we can distribute the work over independent threads, which are then executed on different computing cores. In our model we call this the first level of parallelism. In Sect. 4.2 we describe the basic model that allows us to execute queries in parallel on the

CPU by using independent threads. On the other hand it is possible to use SIMD to exploit the parallel computing capabilities of one CPU core resp. *Streaming Multi Processor* of the GPU. In Sect. 4.3 we make use of this second level of parallelism to extend our model and execute queries efficiently on the GPU. Our GPU framework has some limitations that we discuss in Sect. 4.4.

4.1 JIT-Compilation for SQL Queries

To take full advantage of modern hardware for query processing, we need to use the available processing power carefully. Here, the classical iterator model as well as block-oriented processing have their problems. The iterator model produces a lot of overhead by virtual function calls, while block-oriented processing requires the materialization of intermediate results (see Neuman [14] and Dees et al. [6]). Virtual function calls can cost a lot of processing cycles, similar to materialization, as here we often need to wait for data to be written or loaded. By compiling the whole SQL query into a single function, we can get around virtual function calls and most materialization in order to use processing units as efficiently as possible. For the CPU-only version we used the LLVM framework to compile this function at runtime [6].

Our query execution is fully parallelized since we target modern many-core machines. Neumann [14] already explained that a parallel approach for his code generation can be applied without changing the general patterns. Operations, such as joins and selections, can be executed independently – and therefore in parallel – on different fragments of the data. If other operations are needed we use the patterns described in the next section. Later in Sect. 4.3 we go into details for applying these patterns on GPUs, as here some changes are necessary.

4.2 A Model for Parallel Execution

One of the main challenges when designing parallel algorithms is to keep communication between threads low. Every time we synchronize threads to exchange data, some cores are forced to wait for others to finish their work. This problem can be avoided by partitioning the work in equally-sized chunks and let every thread process one or more chunks independently. In the end, the results of each thread are merged into one final result. This approach is called bulk-synchronous model [16] and describes the first level of parallelism we need to distribute work over independent threads.

Our query execution always starts with a certain table (defined by the join order), which is split horizontally. This is performed only logically and does not imply real work. In the first phase of the query execution every partition of the table is processed by one thread in a tight loop, i.e., selections, joins, and arithmetic terms are executed for each tuple independently before the next tuple is processed. We call this the *Compute* phase, since it is the major part of the work. Before we start the second phase of the query execution, called *Accumulate*, all threads are synchronized once. The Accumulate phase merges the intermediate results into one final result.

This model is similar to the MapReduce model, which is used to distribute work over a heterogeneous cluster of machines [5]. However, our model does not have the limitations of using key-value pairs. Additionally, the Accumulate phase does not necessarily reduce the number of results, while in the MapReduce model the Reduce phase always groups values by their keys. In case of a selection, for instance, the Accumulate phase just concatenates the intermediate results.

The Accumulate phase depends on the type of query, which can be one of the following patterns.

Single Result The query produces just a single result line. For example, this is the case SQL queries with a `sum` clause but no `group by`, see TPC-H query 6:

```
select sum(l_ext...*l_discout) from ...
```

This is the simplest case for parallelization: Each thread just holds one result record. Two interim results can be easily combined to one new result by combining the records according to the aggregation function.

(Shared) Index Hashing On SQL queries with a `group by` clause where we can compute and guarantee a maximum cardinality that is not too large, we can use index hashing. This occurs for example at TPC-H query 1:

```
select sum(...) from ... group by l_returnflag, l_linestatus
```

We store the attributes `l_returnflag` and `l_linestatus` in a dictionary-compressed form. The size of each dictionary is equal to the number of distinct values of the attribute. With this we can directly deduce an upper bound for the cardinality of the combination of both attributes. In our concrete example the cardinality of `l_returnflag` is 2 and `l_linestatus` has 3 distinct values for every scale factor of TPC-H. Therefore the combined cardinality is $2 \cdot 3 = 6$. When this upper bound multiplied with the data size of one record (which consists of several sums in query 1) is smaller than the cache size, we can directly allocate a hash table that can hold the complete result. We use index hashing for the hash table, i.e., we compute one distinct index for the combination of all attributes that serves as a hash key. On the CPU, we differentiate whether the results fit into the L3 (shared by all cores of one CPU) or the L2 (single core) cache. If only the L3 cache is sufficient, we use a shared hash table for each CPU where entries are synchronized with latches. We can avoid this synchronization when the results are small enough to fit into L2 cache. In this case we use a separate hash table for each thread while remaining cache efficient.

For combining two hash tables, we just combine all single entries at the same positions in both hash tables, which is performed similar to the single result's handling. There is no difference in combining shared or non-shared hash tables.

Dynamic Hash Table Sometimes we cannot deduce a reasonable upper bound for the maximum `group by` cardinality or we can not compute a small hash index efficiently from the attributes. This can happen if we do not have any estimation of the cardinality of one of the `group by` attributes or the combination of all `group by` attributes just exceeds a reasonable number. In this case we need a dynamic hash table that can grow over time and use a general 64-bit hash key from all attributes. The following is an example query in which it is difficult to give a good estimation for the `group by` cardinality:

```
select sum() ... from ... group by l_extendedprice,
n_nationkey, p_partsupp, c_custkey, s_suppkey
```

Combining two hash tables is performed by rehashing all elements from one table and inserting it into the other table, i.e., combing the corresponding records.

Array For simple `select` statements without a `group by`, we just need to fill an array with all matching lines. The following SQL query is an example for this problem:

```
select p_name from parts where p_name like '%green%'
```

4.3 Adjusting the Model for GPU Execution

The first adjustment to execute the *Compute/Accumulate* model on the GPU is a technical one: we replace LLVM with OpenCL, which requires a few changes. Before we execute the query, we allocate space for its result in the graphic card's memory, since the kernel is not able to do that on its own. We then call the kernel with the addresses to the required columns, indexes, and the result memory as parameters. After the actual execution the results are copied to the RAM to give the CPU access to it. The result memory on the device can be freed.

Since the GPU is not able to synchronize the workgroups of one kernel, we need to split the two phases into two kernels. As we can see in Fig. 1(a), the workgroups of the *Compute* kernel work independently and write the intermediate results (1) to global memory. The *Accumulate* kernel works only with one workgroup. It writes the final result (3) to global memory (the GPU's RAM).

To differentiate this first level of parallelism from the second, we call this the *global Compute/Accumulate* process. It is a direct translation of the CPU implementation and shows that workgroups on the GPU are comparable to threads on the CPU. Every workgroup has a number of GPU threads, which work in a SIMD fashion. Therefore, the main challenge and the second adjustment is to extend the model by a second level of parallelism to use these threads.

In contrast to CPU threads, every OpenCL workgroup is capable of using up to 1024 *Stream Processors* for calculations. Threads running on those Stream Processors do not run independently but in a SIMD fashion, i.e., at least 32 threads are executing the same instruction at the same time. These threads

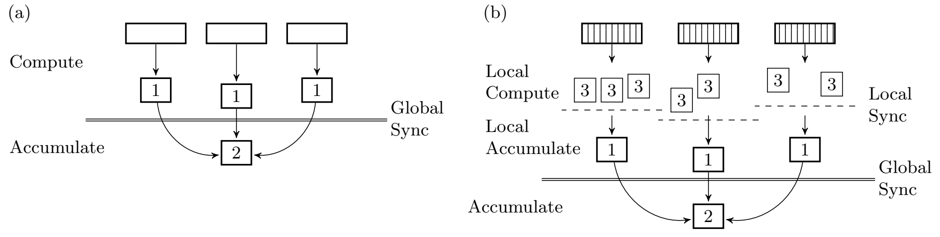

Fig. 1. (a) Basic Model, (b) Extended Model
Legend: 1 – intermediate results, 2 – global result, 3 – private result

form a so called *warp*—the warp size may be different for future architectures. If there are branches in the code, all threads of one warp step through all branches. Threads that are not used in a certain branch execute NOPs as long as the other threads are calculating. The threads of one warp are always in sync, but all threads of one workgroup must be synchronized manually if necessary. Although this local synchronization is cheap compared to a global synchronization, we have to be careful about using it. Every time it is called, typically all workgroups execute it. Since we use up to 1,000 workgroups (see Sect. 5.2), the costs for one synchronization also multiply.

Therefore we apply a *local Compute/Accumulate* model again in the *global* Compute phase. Since the local results are only accessed by the threads of one workgroup, there is no need to write them to global memory. The whole process can be seen in Fig. 1(b). Every thread computes its result in private memory (3), usually the registers of each Stream Processor. At the end of the local Compute phase, the results are written to local memory, the threads are synchronized, and the local Accumulate phase is started. These local processes work in a SIMD fashion and therefore differ from the global phase in the detail.

The local Compute phase works on batches of tuples. The size of the batch equals the number of threads of one workgroup. The thread ID equals the position of the tuple in the batch, i.e., the first thread of the workgroup works on the first tuple, the second on the second, and so on. This works much better than horizontally partitioning the table as we do in the global Compute phase, because we enforce coalesced memory access. The worst case for this SIMD approach arises if a few tuples of the batch require a lot of work and the rest is filtered out early. As we explained above this leads to a lot of cores executing NOPs, while only a few are doing real work. If all tuples of the batch are filtered—or at least all tuples processed by one warp—the GPU can skip the rest of the execution for this batch and proceed to the next. Fortunately this is often the case in typical OLAP scenarios data, e.g., data of a certain time frame is queried and is stored in the same part of the table.

The aggregation specified by the query is in parts already done in the local Compute phase, where each thread works on private memory. In the local/global Accumulate phase we merge these private/intermediate results. The threads are writing simultaneously to local memory, so we require special algorithms to avoid conflicts. The algorithms (prefix sum, sum, merge, and bitonic sort) are well known and can be found in the example directory that is delivered with NVidia's OpenCL implementation. Therefore we just name them at this point and give a short overview but not a detailed description.

The easiest case is no aggregation at all, where we just copy private results to local memory and later to global memory. However, since all threads copy their result in parallel, the right address has to be determined to guarantee that the global result has no gaps. Therefore, each thread counts the number of its private results and writes the size to local memory. To find the right offset for each threads starting position we calculate the prefix sum in local memory. This is done by pairwise adding elements in $\log n$ steps for n being the number of threads. In the end every position holds the number of private results of all threads with smaller IDs and therefore the position in the global result

We use a very similar approach for the `sum`-Aggregation. In case of a single result pattern (see Sect. 4.2) we calculate the sum of the private results in parallel by again using pairwise adding of elements in local memory. For index hashing (see Sect. 4.2) this has to be done for each row of the result.

Sorting is the final process of the Accumulate phase. In every case we can use bitonic sort. For the array pattern it is also possible to pre-sort the intermediate results in the Compute phase and use a simple pair-wise merge in the Accumulate phase.

4.4 Limitations of the GPU Execution

In this section we discuss the limits of our framework. We categorize these in two classes:

Hard limits Due to the nature of our model and the GPU's architecture there are limits that cannot be exceeded with our approach. One of these limits is given by OpenCL: we cannot allocate memory inside kernels, so we have to allocate memory for (intermediate) results in advance. Therefore we have to know the maximum result size and since it is possible that one thread's private result is as big as the final result, we have to allocate this maximum size for every thread. One solution to this problem would be to allocate the maximum result size only once and let every thread write to the next free slot. This method would require a lot of synchronization and/or atomic operations, which is exactly what we want to avoid. With our approach we are limited to a small result size that is known or can be computed in advance, e.g., top-k queries or queries that use `group by` on keys that have relatively small cardinalities. The concrete number of results we can compute, depends on the temporary memory that is needed for the computation, the hardware and the data types used in the result, but it is in the order of several hundred rows.

Soft limits Some use cases are known not to fit to the GPU's architecture. It is possible to handle scenarios of this type on the GPU, but it is unlikely that we can benefit from doing that. The soft limits of our implementation are very common for general-purpose calculations on the GPU. First, the GPU is very bad at handling strings or other variable length data types. They simply do not fit into the processing model and usually require a lot of memory. An indication for this problem is, that even after a decade of research on general-purpose computing on graphics processing units and commercial interest in it, there is no string library available for the GPU, not even simple methods such as `strstr()` for string comparison. In most cases we can avoid handling them on the GPU by using dictionary encoding. Second, the size of the input data must exceed a certain minimum to use the full parallelism of the GPU (see Sect. 5.2).

Furthermore, GPUs cannot synchronize simultaneously running workgroups. Therefore we cannot efficiently use algorithms accessing shared data structures, such as the shared index hashing and the dynamic hash table approach described in Sect. 4.2.

5 Evaluation

In this section we present the performance results of our framework. We compare the performance of NVidia's Tesla C2050 to a HP Z600 workstation with two Intel Xeon X5650 CPUs and 24 GB of RAM. The Tesla GPU consists of 14 Streaming Multiprocessors with 32 CUDA cores each and has 4 GB of RAM. The JIT compilation was done by LLVM 2.9 for the CPU and the OpenCL implementation that is shipped with CUDA 5.0 for the GPU.

5.1 GPU and CPU Performance

Since the native CPU implementation described in Sect. 4 was already compared to other DBMS [6], we take it as our baseline and compare it to our OpenCL implementation. In our first experiment we compare the implementations by measuring the raw execution time of each query. As explained in Sect. 3.2 the necessary columns for execution are in memory, we do not consider the compile times, and we include result memory allocation (CPU and GPU) and result transfer (only needed on GPU). We configure the OpenCL framework to use 300 workgroups with 512 threads each on the GPU and 1000 workgroups with 512 threads on the CPU. This configuration gives good results for all queries as we show in Sect. 5.2.

Figure 2 compares the execution times for seven TPC-H queries on different platforms: Z600 is the native CPU implementation on the Z600 machine, Tesla/CL the OpenCL implementation on the Tesla GPU and Z600/CL the OpenCL implementation executed on the CPU of the workstation. As we can see, the OpenCL implementation for the GPU is considerably faster than the native implementation on the workstation for most queries. The only exception is Q1, which takes almost twice as long. The OpenCL implementation of Q1 on the CPU is significantly worse than native implementation as well. The reason for this is hard to find, since we cannot measure time within the OpenCL kernel. The performance results might be worse because in contrast to the other

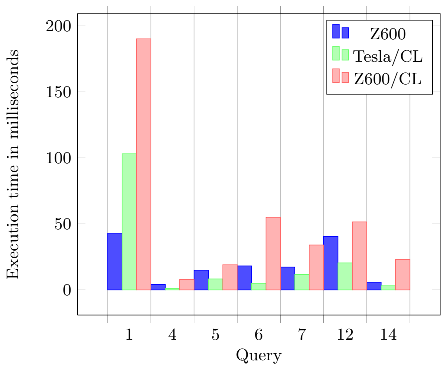

Fig. 2. Comparison of execution times

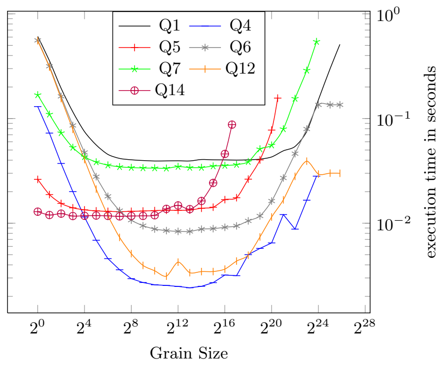

Fig. 3. Comparison of different grain sizes

queries the execution of Q1 is dominated by aggregating the results in the local accumulate phase. This indicates that the CPU works better for queries without joins but with aggregations. Further tests and micro-benchmarks are needed to proof this hypothesis.

It is remarkable that the OpenCL implementation on the CPU achieves almost the same performance as the native CPU implementation for half of the queries, considering that we spent no effort in optimizing it for the actual architecture.

5.2 Number of Workgroups and Threads

We illustrate in Fig. 4 how the number of workgroups used for the execution of the query influences the performance. We measure two different configurations: the left figure shows the results when we use 512 threads per workgroup; on the right we use 256 threads per workgroup. Due to implementation details our framework is not able to execute queries 1 and 5 for less than around 140 resp. 200 workgroups. For the other queries we can clearly see in both figures that we need at least around 50,000 threads in total to achieve the best performance. This high number gives the task scheduler on the GPU the ability to execute instructions on all cores (448 on the Tesla C2050) while some threads are waiting for memory access. There is no noticeable overhead if more threads are started with the following exceptions:

Query 4 shows irregular behavior. It is the fastest query in our set, the total execution time is between 1 and 2 ms. Therefore uneven data distribution has a high impact.

Query 5 is getting slower with more threads, because the table chunks are too small and therefore the work done by a thread is not enough. The table, which is distributed over the workgroups, has only 1.5 mio rows, i.e., each thread processes only 3 rows in case of 1,000 workgroups with 512 threads each. This matter is even worse with Query 7, because the table we split has only 100,000 rows. The other queries process tables which have at least 15 mio rows.

This experiment shows that the GPU works well with a very high number of threads as long as there is enough work to distribute. In our experiments, one thread should process at least a dozen rows.

The behavior is similar to task scheduling on the CPU, where task creation does not induce significant overhead in contrast to thread creation. Figure 3 shows the execution time of the native implementation depending on the *grain sizes*, which is the TBB term for the size of the partition and therefore controls the number of tasks. As we can see, a large grain size prevents the task scheduler from using more than one thread and therefore slows down execution. The optimal grain size for all queries is around 1000. We achieve a speed-up of 7 up to 15 for this grain size. If the partitions are smaller, the execution time increases again. Similar to our OpenCL implementation the optimal number of tasks per core is around 100.

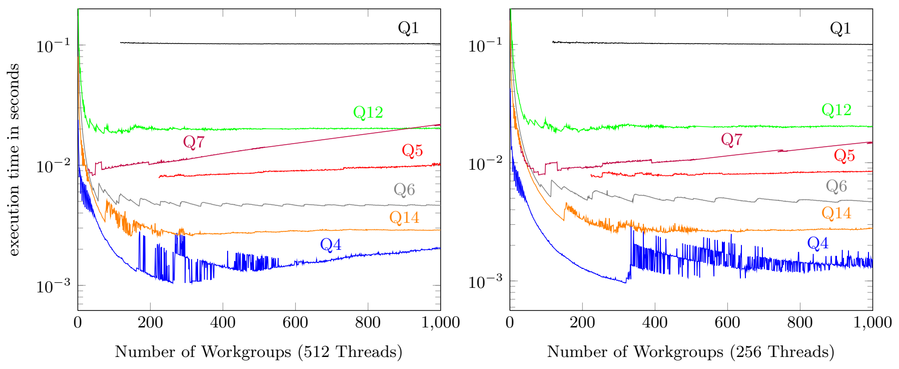

Fig. 4. Comparison of different workgroup numbers (left: 512, right: 256 threads per workgroup)

6 Conclusion

In this work, we have investigated the parallel execution of OLAP queries on massively parallel processor architectures by leveraging a JIT query compilation approach. Our goal was to identify and evaluate patterns for parallelization that are suitable both for CPUs and GPUs. As a core pattern we have presented a two-phase Compute/Accumulate model similar to the MapReduce paradigm, which requires only one global synchronisation point. For compiling the whole query into executable code we use the OpenCL framework which already provides a JIT compiler for different devices. Surprisingly we were able to execute our GPU-optimized code on the CPU as well. Although, this is still slower than a native CPU optimized implementation, the advantage of having only one implementation for different platforms may be worth the loss in performance. Since

OpenCL is an open standard for parallel platforms, we expect that our framework also runs on other coprocessors.

Our experimental results using a subset of TPC-H have shown that GPUs can be used to execute complete OLAP queries and even can outperform modern CPUs for certain queries. However, we have also discussed some limitations and differences between CPU and GPU processing which have to be taken into account for query compilation and execution.

References

1. Bakkum, P., Skadron, K.: Accelerating SQL database operations on a GPU with CUDA. In: Proc. 3rd Workshop on GPGPU. p. 94 (2010)
2. Boncz, P., Zukowski, M., Nes, N.: MonetDB/X100: Hyper-Pipelining Query Execution. In: Proc. CIDR. vol. 5 (2005)
3. Boncz, P.A., Kersten, M.L., Manegold, S.: Breaking the memory wall in monetdb. Commun. ACM 51(12), 77–85 (2008)
4. Daga, M., Aji, A., Feng, W.: On the Efficacy of a Fused CPU+GPU Processor (or APU) for Parallel Computing. In: SAAHPC. IEEE (2011)
5. Dean, J., Ghemawat, S.: MapReduce. Comm. of the ACM 51(1), 107 (Jan 2008)
6. Dees, J., Sanders, P.: Efficient Many-Core Query Execution in Main Memory Column-Stores. to appear on ICDE 2013 (2013)
7. Graefe, G.: Encapsulation of Parallelism in the Volcano Query Processing System, vol. 19. ACM (1990)
8. He, B., Lu, M., Yang, K., Fang, R., Govindaraju, N.K., Luo, Q., Sander, P.V.: Relational Query Coprocessing on Graphics Processors. ACM Transactions on Database Systems 34 (2009)
9. He, B., Yu, J.X.: High-Throughput Transaction Executions on Graphics Processors. PVLDB 4, 314–325 (Mar 2011)
10. Kaldewey, T., Lohman, G., Mueller, R., Volk, P.: GPU Join Processing Revisited. Proc. 8th DaMoN (2012)
11. Kemper, A., Neumann, T.: HyPer: A hybrid OLTP&OLAP Main Memory Database System Based on Virtual Memory Snapshots. In: Proc. of ICDE (2011)
12. Krikellas, K., Viglas, S., Cintra, M.: Generating Code for Holistic Query Evaluation. In: Proc. of ICDE. pp. 613–624. IEEE (2010)
13. Lattner, C.: LLVM and Clang: Next Generation Compiler Technology. In: The BSD Conference (2008)
14. Neumann, T.: Efficiently Compiling Efficient Query Plans for Modern Hardware. Proc. of VLDB 4(9), 539–550 (2011)
15. NVidia: CUDA C Best Practices Guide (2012)
16. Valiant, L.G.: A Bridging Model for Parallel Computation. Comm. ACM 33 (1990)
17. Zukowski, M., Boncz, P.A.: From X100 to Vectorwise: Opportunities, Challenges and Things Most Researchers do not Think About. In: SIGMOD Conference (2012)