Like Water and Oil: With a Proper Emulsifier, Query Compilation and Data Parallelism Will Mix Well

Henning Funke
DBIS Group, TU Dortmund University
henning.funke@cs.tu-dortmund.de

Jens Teubner
DBIS Group, TU Dortmund University
jens.teubner@cs.tu-dortmund.de

1 INTRODUCTION

In response to physical limitations, hardware has changed significantly during the past two decades. As the database community, we have no chance but adapt to those changes in order to benefit from these and further hardware advances.

Two strategies to deal with the change have proven particularly successful. To avoid hitting the *memory wall*, modern engines *compile queries* into native machine code [**?**]; this way, data can be kept longer in registers and performance-limiting memory I/Os can be avoided. To escape the *power wall*, the use of heterogeneous and massively parallel architectures has been proposed; *graphics processors (GPUs)* in particular can deliver spectacular compute performance at a very attractive power footprint. But while both these strategies are very successful and well understood, it is surprisingly difficult to bring both together without losing much of their benefit.

In this demo, we showcase *DogQC*, the query compiler that we develop at TU Dortmund University. DogQC includes the *Lane Refill* and *Push-Down Parallelism* techniques to combat *divergence effects* that are the root cause for the above mentioned difficulty. The two techniques very effectively avoid resource under-utilization on graphics processors, while leveraging the bandwidth efficiency of compiled code. In practice, DogQC's anti-divergence measures can improve query performance by several factors.

PVLDB Reference Format:
Henning Funke and Jens Teubner. Like Water and Oil: With a Proper Emulsifier, Query Compilation and Data Parallelism Will Mix Well. PVLDB, 13(12): 2849–2852, 2020.
doi:https://doi.org/10.14778/3415478.3415491

1.1 Divergence in GPU-Based Execution

The root cause for the discrepancy between query compilation and (heterogeneous) parallelism is *divergence*. To understand the effect, consider the plan excerpt from TPC-H Q10 shown here as Fig. 1 on the right. A query compiler will attempt to compile the plan region into a straight-line sequence of code, a *pipeline*. The motivation to do so is to propagate tuples within registers, rather than spilling data to (slow) memory.

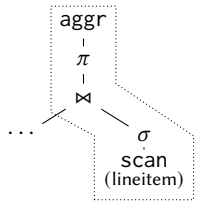


Figure 1: Plan excerpt.

Proceedings of the VLDB Endowment, Vol. 13, No. 12 ISSN 2150-8097.
doi:https://doi.org/10.14778/3415478.3415491

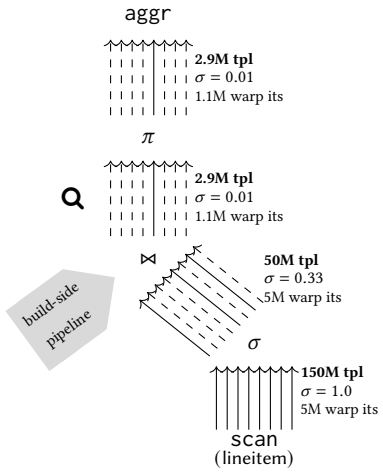


Figure 2: GPU under-utilization due to filter divergence.

During execution, not all lineitem tuples will actually traverse the full pipeline. Some tuples might instead be *eliminated* by operators such as filter σ or join $\bowtie$. If this happens, a sequential processor will immediately abort the pipeline, continue with the next input item, and hence keep CPU efficiency at peak.

Data-parallel execution back-ends, by contrast, do not have the option of aborting a pipeline early, unless *all* tuples in the same batch of work are eliminated.

Figure 2 illustrates this effect for a GPU-based back-end (assuming a batch—or *"warp"*—size of eight for illustration purposes). In some warp iteration, only *warp lanes* 1, 5, and 7 might have passed the filter σ, leaving the five remaining warp lanes *inactive* (indicated as dashed arrows - ->). The following join de-activates another two warp lanes, bringing GPU efficiency down to 1/8 in this example.

The resulting GPU under-utilization is even worse in real settings. To scan a lineitem table with 150 million rows, actual GPUs will require 5 million *warp iterations*, each consisting of 32 warp lanes. Although σ filters out about 2/3 of all rows, it is extremely unlikely that all lanes within a warp become inactive. Therefore, (almost) all 5 million warp iterations proceed into the join operator $\bowtie$. Only 1 % of the remaining rows find a match during the join. In an actual data set, 2.9 million rows remain after the join, but they are spread across 1.1 million warp iterations. Ideally, the projection π and aggregation aggr operators could have been processed by only 2.9 M/32 = 90 K warp iterations. In other words, state-of-the-art query compilation techniques will leave 92 % of the GPU's processing capacity unused.

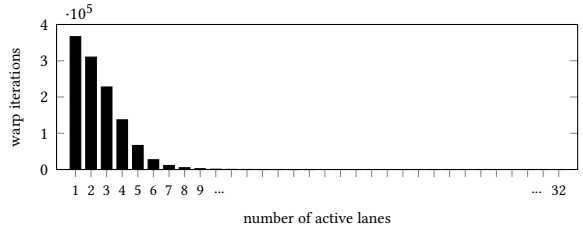


Figure 3: Lane activity profile with filter divergence.

1.2 GPU Query Compiler DogQC

GPU code generated by our query compiler *DogQC*[1] leverages *Lane Refill* and *Push-Down Parallelism* techniques to counter divergence effects like the ones we described. In the rest of this demonstration proposal, we will give a high-level idea of the *Lane Refill* and *Push-Down Parallelism* techniques (Sections 2 and 3), then report on experimental results for DogQC (Section 4). In Section 5, we describe how we intend to demonstrate the internals of the DogQC engine at VLDB before we wrap up in Section 6. More details on the *Lane Refill* and *Push-Down Parallelism* mechanisms can be found in the respective full paper [**?**].

2 *LANE REFILL* TECHNIQUE

Divergence effects (here: *filter divergence*) are a consequence of the SIMT, "single instruction, multiple threads," execution paradigm embodied in all modern graphics processors. A number of threads (or *lanes*, typically 32 of them) is grouped into a *warp*. During execution, *all* lanes within a warp execute the *same* GPU instruction.

The SIMT model encounters a problem whenever some lanes or data elements need a different amount or kind of processing than others. In such situations, control flows will *diverge*. Since all lanes within a warp *still* execute the same instruction, lanes will be turned *inactive* and their computation result will be discarded. As illustrated above, this can result in resource under-utilization.

To illustrate the severity of this effect, we instrumented the query plan shown earlier (Figure 2) to monitor warp utilization at the plan point marked with a magnifying glass 🔍. Figure 3 shows a histogram on the number of warps that have passed this plan stage with a warp utilization of 1, …, 32 active lanes. It is easy to see that only a fraction of the available compute capacity is used; in most warps, only one or two out of 32 warp lanes performed actual work.

2.1 Balance Operators and Refill Buffers

To combat the situation, DogQC injects *balance operators* into the relational query plan. Code generated for these operators detects warp under-utilization at runtime. Whenever utilization drops below a configured threshold, the state of all remaining active lanes is suspended to a *refill buffer* and the pipeline starts over with a fresh set of input tuples.

Figure 4 illustrates this for three successive warp iterations ① through ③. Since only 2, 1, and 3 lanes remained active in these iterations (respectively), their state is flushed to the refill buffer.

[1] https://github.com/Henning1/dogqc

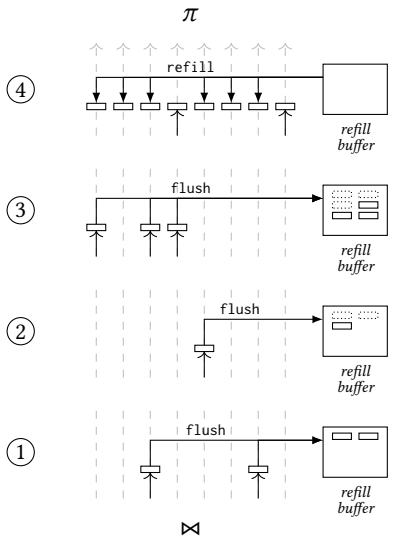


Figure 4: *Lane Refill*: tuples from three low-activity iterations are suspended to the *refill buffer* and resumed for full lane activity in the fourth iteration.

After flushing, each of those warp iterations is terminated and processing starts over with the next set of input tuples.

2.2 Refilling

As soon as a sufficient number of lane states has been stored to the refill buffer, the buffer can be used to *refill* lanes that have become inactive. This time, the under-utilized warp iteration is not terminated but continues processing with full utilization after refilling. This is visualized in Step ④ of Figure 4. Here, only two out of eight warp lanes remained active after the downstream join operator. Using the refill buffer, the remaining six warp lanes can be filled with useful work, resulting in full warp utilization upstream.

Implementation-wise, flushing and refilling are backed up in DogQC by CUDA's `__ballot_sync`, `__popc` ("population count"), and shuffling primitives. These primitives are highly efficient; balance operators will cause little overhead even when only few warps go below the utilization threshold.

2.3 Effect of *Lane Refill*

Lane Refill brings warp utilization back to a high compute efficiency. Following the balancing operator, all executed warps (except for the last warp in each grid block) are *guaranteed* to have a warp utilization above the configured threshold.

In Figure 5, this is illustrated with a histogram for the same plan point that we profiled earlier (Figure 3), but this time with a balance operator applied. The histogram confirms that *(a)* (almost) no warps exist with a utilization below 26 lanes (the threshold we configured); and *(b)* the total number of executed warps has dropped by a factor of about ten. In terms of overall execution performance, *lane refill* will improve execution times by about 2–3x for the example plan shown in Figure 2.

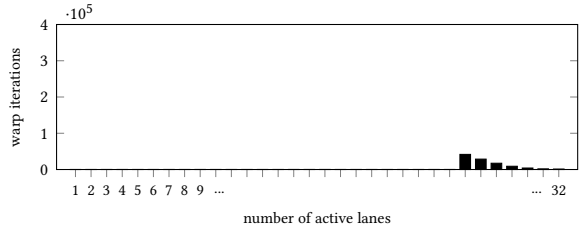


Figure 5: Lane activity profile with lane refill buffer to consolidate filter divergence.

3 *PUSH-DOWN PARALLELISM*

DogQC's *Push-Down Parallelism* technique addresses another flavor of divergence that may arise orthogonally to the aforementioned filter divergence. *Expansion divergence* is the effect when a different amount of work is needed to process each of the items within a warp. Database *join operations* are a common situation where this effect arises.

Figure 6 here on the right illustrates the effect. Probe side tuples coming from the right may find a different number of join partners each. Specifically, in the example, lane 6 will have significantly more tuples to process than the remaining warp lanes. In such a situation, existing query compilers will process all matches of a single probe-side tuple *within* the same warp lane. In the example, execution times would be dominated by the sequential processing of all matches for lane 6.

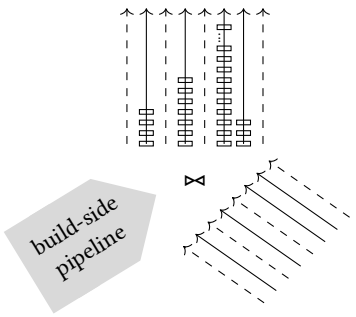


Figure 6: Expansion divergence.

Push-Down Parallelism—detailed in [**?**]—mitigates the situation by parallelizing the processing the matches of a single probe-side tuple *across* the available warp lanes. To this end, the execution state of probe-side lanes is *broadcast* over lanes, while build-side matches are *partitioned* across. Again, we leverage efficient CUDA primitives, such as `__ballot_sync` and `__shfl_sync` ("shuffle sync").

As illustrated in Figures 7 and 8, *Push-Down Parallelism* improves lane utilization and reduces the overall number of iterations needed to complete the query.

Lane Refill and *Push-Down Parallelism* complement one another, and Figure 6 shows an example where both flavors of divergence co-exist. Another typical occurrence of expansion divergence is the processing of *variable-length data*, strings in particular. If possible, DogQC will parallelize the processing of strings across warp lanes to improve resource utilization.

4 EVALUATION

With *DogQC*, we provide a query compiler with a wide range of SQL functionality; sufficient to support all queries from the TPC-H benchmark set.

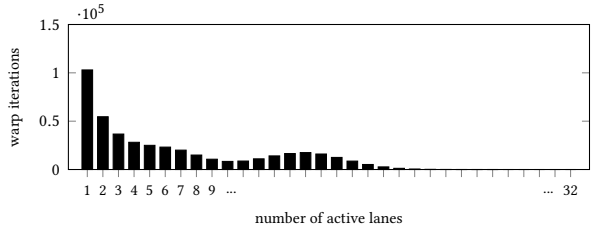


Figure 7: Lane activity with expansion divergence.

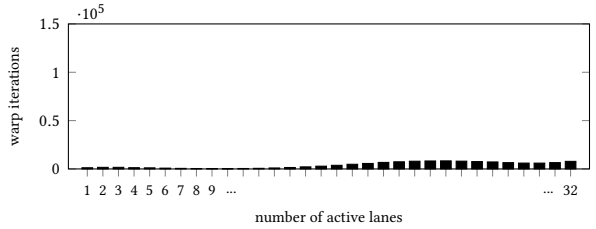


Figure 8: Lane activity profile with push-down parallelism to consolidate expansion divergence.

4.1 TPC-H Performance

To assess the benefits of measures to contain divergence, we performed a series of measurements with the TPC-H benchmark set. Our measurements were based on an NVIDIA RTX2080 GPU with 46 Streaming Multiprocessors and 8 GB GPU memory, installed in a host system with an Intel i7-9800X GPU and 32 GB of main memory. As a reference, we compared DogQC with the hybrid CPU/GPU system *OmniSci* [**?**].

Our benchmark results are depicted in Figure 9. For each of the 22 TPC-H queries, the bars indicate query execution time assuming that the data set is resident in GPU memory.

For OmniSci, we report the total wall clock time needed to execute the query as well as the amount of time spent on GPU processing. OmniSci is a hybrid execution engine, meaning that both, CPU and GPU, will be used to jointly answer the query. As can be seen in the figure, several queries can, in fact, not benefit much from GPU acceleration in OmniSci. Also mind that OmniSci could successfully execute only 13 of the 22 TPC-H benchmark queries.

The focus of this demonstration is on avoiding divergence effects. To this end, we prepared a version of DogQC where the divergence-related optimizations can be turned off (if appropriate, see below). In the graph, this is reported as "naive." As can be seen in the figure, the mitigation of divergence will result in a significant performance improvement for some queries, while never having any negative impact on any query. DogQC can run all 22 TPC-H queries entirely on the GPU (benefits from hybrid CPU/GPU processing would be orthogonal to divergence mitigation).

A secondary benefit of divergence handling in DogQC cannot directly be observed in the figure. An important flavor of divergence stems from the processing of (variable-length) *strings*. Existing systems, including OmniSci, circumvent the problem and apply *dictionary encoding* on all string data. The resulting overhead on *ingestion speed* and *memory requirement* cannot be inferred from

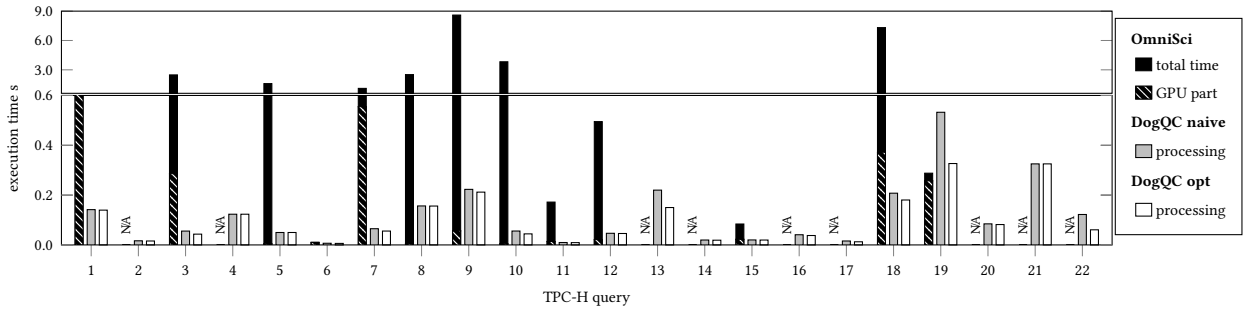


Figure 9: Execution times of DogQC for TPC-H benchmark queries (scale factor 25). The divergence optimizations improve query performance.

Figure 9. DogQC, by contrast, can naturally handle variable-length data, including strings (also in its "naive" configuration). See [**?**] for details.

5 DEMO SETUP

Our demonstration at VLDB will enable visitors to look under the hoods of the DogQC query compiler, with a focus on anti-divergence techniques.

DogQC provides mechanisms to visualize generated query plans (we leverage the `dot`[2] utility for this purpose), which demo spectators can use for inspection. An example of an actual query plan is shown in Figure 10(a) for the TPC-H Q10 plan sketched in Figure 2. As part of the demo, visitors will be able to freely place balance operators into DogQC-generated query plans and observe their effects (in Figure 10(b), a balance operator—highlighted in red—has been injected, corresponding to the 🔍 marker in Figure 2).

Balance operators (if placed properly) will have an immediate effect on query execution speeds, which demo visitors will be able to verify with TPC-H and other data sets.

To inspect the inner workings of anti-divergence techniques, DogQC is equipped with profiling mechanisms that visualize GPU lane utilization. At the demo, visitors will be able to generate histogram graphs like those in Figures 3 and 5 for their own queries and at arbitrary points in the query plan.

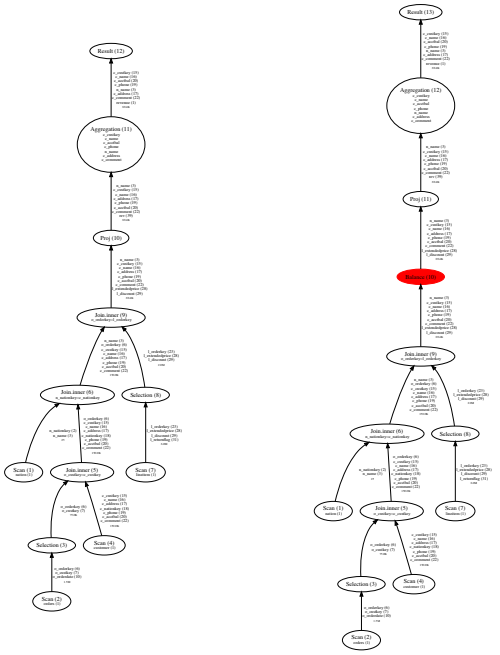
Finally, demo visitors will be able to verify the utilization of further GPU resources, such as registers, memories, or caches.

6 SUMMARY

Divergence effects can seriously impair the performance potential of modern, data-parallel execution platforms such as GPUs. With help of the *Lane Refill* and *Push-Down Parallelism* techniques, our query compiler *DogQC* can combat divergence effects and restore processing efficiency.

DogQC supports the full TPC-H benchmark set. In the demo, visitors will be able to experiment with DogQC, state their own queries, and watch the inner workings of DogQC.

[2]https://www.graphviz.org/



(a) "naive" query plan (b) with balance operator

Figure 10: DogQC query plans corresponding to the TPC-H Q10 plan sketched in Figure 2. Left: query plan without balance operators; right: plan with balance operator injected after the join operator (corresponding to 🔍 in Figure 2).

ACKNOWLEDGEMENTS

This work was supported by the DFG, Collaborative Research Center SFB 876, project A2, and DFG Priority Program "Scalable Data Management for Future Hardware" (TE1117/2-1).

REFERENCES

[1] H. Funke and J. Teubner. Data-parallel query processing on non-uniform data. *PVLDB*, 13(6):884–897, 2020.
[2] T. Neumann. Efficiently compiling efficient query plans for modern hardware. *PVLDB*, 4(9):539–550, 2011.
[3] OmniSci Incorporated. OmniSciDB. `https://www.omnisci.com/`, 2019.