Data-Parallel Query Processing on Non-Uniform Data

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

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

ABSTRACT

Graphics processing units (GPUs) promise spectacular performance advantages when used as database coprocessors. Their massive compute capacity, however, is often hampered by *control flow divergence* caused by non-uniform data distributions. When data-parallel work items demand for different amounts or types of processing, instructions execute with lowered efficiency. *Query compilation* techniques—a recent advance in GPU-accelerated database processing—suffer from the problem even more, because divergence effects are amplified during the execution of fused pipeline operators.

In this work, we identify two types of control flow divergence—*filter divergence* and *expansion divergence*—that frequently occur in real world workloads. We quantify the problem for two poster cases and propose techniques to balance these divergence effects. By balancing divergence effects, our approach is able to restore processing efficiency even when pipelines contain heavily skewed operations. Our query compiler DogQC has a wider range of functionality than other query coprocessors *and* achieves performance improvements. We observe shorter execution times for TPC-H benchmark queries by factors up to 4.51x compared with existing GPU query compilers and by factors up to 4.54x compared with CPU-based systems.

PVLDB Reference Format:
Henning Funke, Jens Teubner. Data-Parallel Query Processing on Non-Uniform Data. *PVLDB*, 13(6): 884-897, 2020.
DOI: https://doi.org/10.14778/3380750.3380758

1. INTRODUCTION

Data-parallelism is frequently used for efficient query processing (e.g. SIMD, coprocessors). As means of specialization, it is a way to overcome the *power wall* that limits the design of modern multiprocessors [6]. Instead of dedicating chip resources to control flow management, data-parallel architectures target throughput. For instance, executing an instruction for 32 fields at a time reduces control flow management work by 32x compared to scalar execution.

Proceedings of the VLDB Endowment, Vol. 13, No. 6
ISSN 2150-8097.
DOI: https://doi.org/10.14778/3380750.3380758

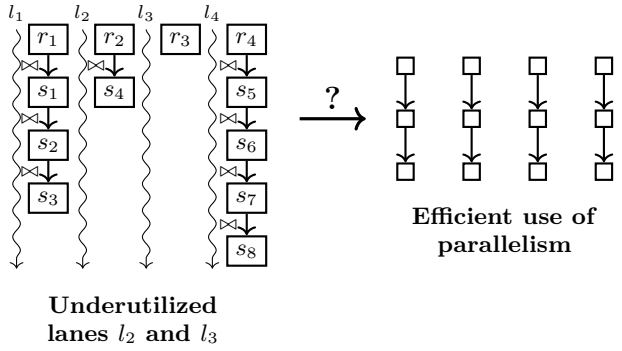

Figure 1: Data-parallel computation of $R \bowtie S$ with inefficient use of compute resources due to non-uniform distribution of S.

Leveraging data-parallelism in a beneficial way can be challenging. While uniform data can be processed naturally, irregular data and computation patterns may compromise the benefits. In the uniform case, it is sufficient to package data into parallel lanes and then to run an instruction sequence. Non-uniform data, however, cannot easily be packaged into a fixed number of fields and the instruction sequences may *diverge*. Consequently, for irregular problems, data-parallel operations execute with lowered efficiency.

Figure 1 illustrates the problem for a database join operation. While rows r_1 and r_4 find three/four join partners, there is only a single join partner for r_2 and none for r_3. A naive data-parallel execution, therefore, will leave execution lanes l_2 and l_3 underutilized.

In real-world problems, unfortunately, such irregularities are the norm, rather than the exception, e.g.

Variable Length Data. The size of an attribute may vary across different entities (e.g. strings).

Skewed Distributions. Skewed data distributions lead to divergence during recombination tasks (e.g. joins).

Computation Divergence. As a secondary effect of data properties, divergence may occur during computations (e.g. hash collisions).

1.1 State of the Art

Non-uniformness can be particularly harmful to parallel *query compilation* approaches. Query compilation closely entwines sequences of operators (*pipelines*) into native code.

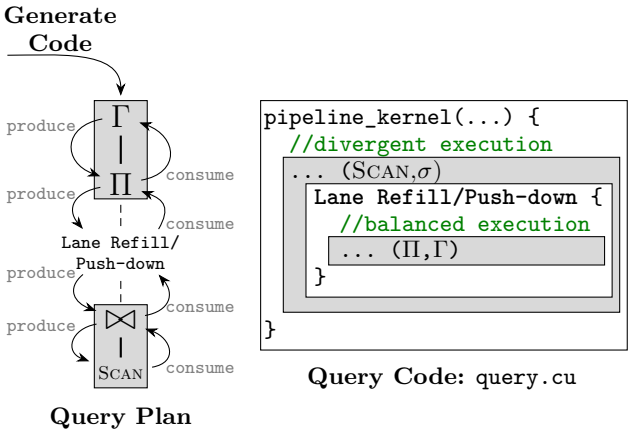

Figure 2: Injecting divergence balancing into query code generation.

Thus non-uniform effects that occur in the data-parallel execution of one operator may be amplified during the execution of successive operators. In CPU-based systems, the problem of data-parallelism in compiled pipelines has been addressed by database researchers [20, 34]. A promising approach by Lang et al. [19] refills inactive SIMD lanes with buffered elements from previous low-activity iterations.

By contrast, in the context of data-parallel accelerators—such as GPUs—existing systems tend to circumvent the problem of non-uniformity at a high price. E.g., they use string dictionaries [24, 3, 7, 13, 12], specialized joins [30], materialization barriers [37, 13, 7], or bit-packed keys [7, 8] to provide a uniform surrogate. The surrogate, however, usually has limited expressivity, and query coprocessing engines struggle to match the same range of operations supported by their CPU counterparts.

1.2 System: DogQC

We use our query compiler *DogQC* to illustrate our techniques to cope with non-uniformity on data parallel processing devices. DogQC performs *Just-in-Time compilation* (JIT) with standard template-based code generation [25], which we apply to GPUs with the techniques established in HorseQC [12]. The mechanisms to cope with non-uniformity are *orthogonal* to other GPU-based query processors.

The approach is illustrated in Figure 2, which shows an operator with `Lane Refill/Push-down` for divergence balancing that is placed between standard relational operators (shown in gray). During JIT-compilation the query plan on the left is translated to the query code on the right. For the balancing operator, DogQC instantiates a code template that is weaved into the code for relational processing. The mechanism resolves imbalances in the code of the operators with divergence effects (outer gray box) before continuing with the execution of succeeding operators (inner gray box). In this way, the succeeding operators are executed with increased processing efficiency.

By balancing divergence effects, DogQC targets efficient query processing without assumptions about the uniformness of workloads. This allows DogQC to achieve a larger range of functionality and to avoid expensive preprocessing steps that are typically used to harmonize data.

1.3 Contributions and Outline

Our work is the first to pinpoint the problem of divergence in the context of GPU-accelerated database processing (Section 2). We identify two flavors of divergence: *expansion divergence* (Section 3) and *filter divergence* (Section 4). With *Push-down Parallelism* (Section 3.2) and *Lane Refill* (Section 4.2), we provide novel and effective mechanisms to counter the two divergence effects. In an extensive set of experiments (Section 5), we demonstrate how Push-down Parallelism and Lane Refill can speed up query processing by more than a factor of two for realistic benchmarks.

To round up this report, we discuss related work in Section 6, and summarize in Section 7.

2. NON-UNIFORM PIPELINES

Data-parallel processing of non-uniform data encounters the following problem: Some data elements need a different amount or kind of processing than others. Consequently, parallel lanes need to diverge to follow their tuples' processing path. Due to this effect, called *control flow divergence*, (short: divergence) the affected lanes may idle or unmatched execution paths are sequentialized. The advantage of data-parallelism to reduce the amount of control flow work is compromised.

Control flow divergence is particularly harmful in kernel-programs[1] that execute operator sequences (e.g. $\mathsf{op}_1 \ldots \mathsf{op}_n$) as they are typical in compiled query pipelines [12]. If the operator op_i introduces divergence, the subsequent operators $\mathsf{op}_{i+1} \ldots \mathsf{op}_n$ may suffer from it as well. For example, a tuple that is filtered out should be disregarded by the following operators, leaving the respective lane idle throughout.

In the following, we take the TPC-H benchmark and analyze the divergence effects that occur in actual query pipelines. We differentiate between two types of control flow divergence, called *filter divergence* and *expansion divergence*. Their difference is based on properties of the operation they originate from.

2.1 Lane Activity

Data-parallel processors execute instructions on multiple lanes at a time, e.g. GPUs execute instructions in warps of 32 lanes. Starting with scan, each warp reads the attribute data for 32 tuples into an on-chip register file [14, 27]. Each of the warp lanes is responsible for one scanned tuple and we call the lane *active* when it holds at least one tuple to pass on to the next operator. In subsequent operators, lanes may resign from their tuple, e.g. by applying a filter. However, warp instructions will still compute a value for these *passive* lanes, but the result is discarded. Passive lanes do not contribute to the computation, but cause dissipation of chip resources for register allocation and instruction execution. To achieve a high execution efficiency, it is important to minimize the number of passive lanes.

3. EXPANSION DIVERGENCE

Expansion divergence occurs in operators such as string comparisons and joins, where parallel lanes need to process varying amounts of work items depending on data properties. Expansion divergence can lower the execution-efficiency due to divergence in the operator itself (e.g. comparisons of short

[1]Parallel GPU procedures, called kernels in short.

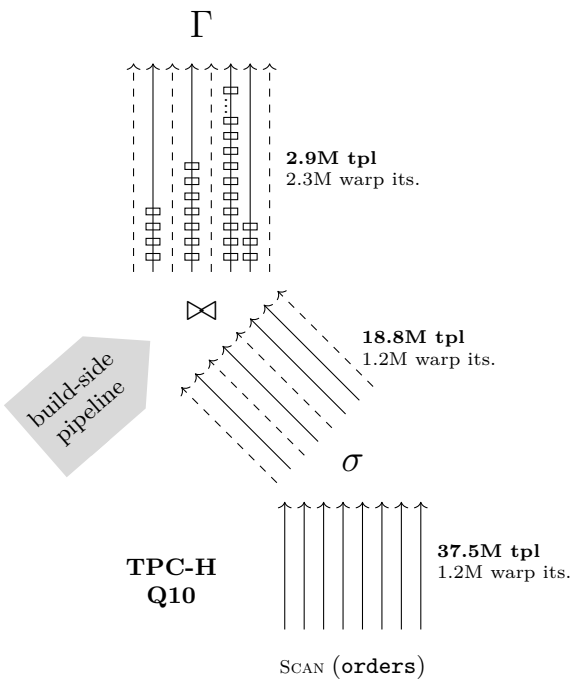

Figure 3: Analytic benchmark query with expansion divergence in join operator. Varying numbers of join matches cause more warp iterations for fewer tuples.

strings finish early) and due to divergence in subsequent operators. The latter occurs when the expansion process creates a varying amount of new tuples, e.g. join matches.

3.1 Poster Case 1

TPC-H query 10 contains a join between the `orders` and `lineitem` tables. Both tables are filtered, therefore optimizers may decide on `orders` ⋈ `lineitem` or `lineitem` ⋈ `orders`. For the latter DogQC computes a hash join with `lineitem` as build relation and `orders` as probe relation. During probe, the tuples from `orders` have varying numbers of matches, which correspond to the items in an order. Producing the matches is a process with expansion divergence. To analyze the execution efficiency, we execute the query with DogQC and look at two metrics at each pipeline stage: The number of tuples and the number of *warp iterations*. The number of warp iterations indicates how many times a warp of 32 lanes goes through an operation. If at least one element is active, the full warp performs the iteration. However, each iteration can process up to 32 elements.

Figure 3 illustrates the compiled pipeline[2]. First, a scan of 37.5 M tuples from `orders`, then selection leaving 18.8 M tuples active, and then join probe producing 2.9 M match tuples. The scanned `orders`-tuples are evenly parallelized and thus processed in 37.5 M/32 ≈ 1.2 M warp iterations. Selection has the same number of warp iterations because almost all warps have remaining tuples. The following join probe produces a lower number of 2.9 M tuples but requires a higher number of 2.3 M warp iterations. Each lane iterates through varying match numbers and only 2.9 M/2.3 M ≈ 1.3

[2]Figures 3,4,6, and 7 use a lower number of 8 warp lanes for illustration purposes. The actual hardware in this work uses 32 warp lanes.

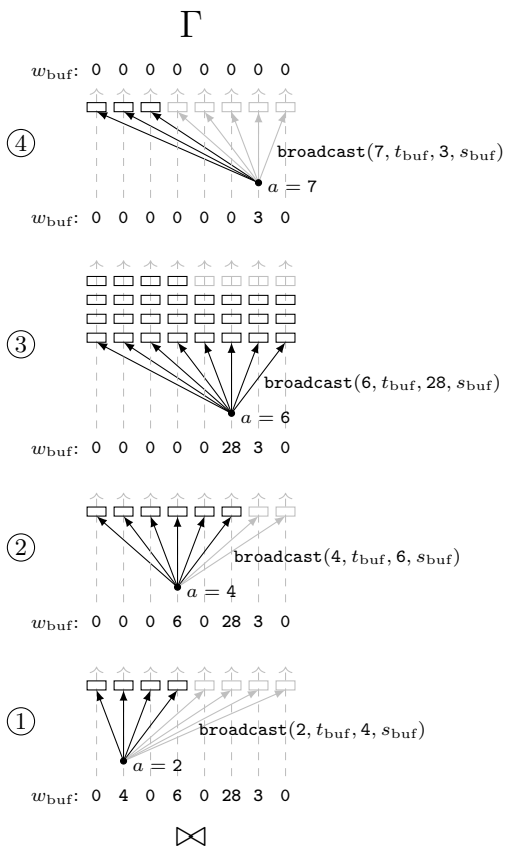

Figure 4: Illustration of Push-down Parallelism that expands the join matches of four warp lanes.

lanes per warp are active on average. In an ideal setting only 2.9 M/32 ≈ 0.1 M warp iterations would be sufficient. Expansion divergence that occurs in the join probe operator causes a low execution efficiency.

3.2 Push-down Parallelism

Existing query compilers [20, 8, 12] parallelize over the scanned table. *Within* each parallelization unit, expansion processes are executed *sequentially*. For example in the join $R \bowtie S$, where $r \in R$ is part of the scanned table, all join matches of r with S are produced by the same thread. This causes inefficiency as lanes diverge along the distribution of join matches. In the worst case the operators op_i to op_n are executed sequentially when all tuples with matches are processed by the same lane.

Push-down Parallelism has the ability to prevent this effect by changing the parallelization strategy *within the pipeline*. For operators with expansion properties, it pushes parallelization down one level to the expansion process. E.g. for joins, the parallelization level moves from parallelizing over the scanned tuples of R to parallelizing over the join matches with S. This is achieved with *broadcast operations* that redistribute parallel work.

We describe how Push-down Parallelism redistributes work to prevent imbalance caused by join expansion. Figure 4 illustrates this and we formalize the mechanism as pseudocode in Figure 5. Before applying Push-down Parallelism, warps have gone through the previous operators op_1 to op_{i-1} (lines 1–5).

```
PUSH-DOWN PARALLELISM
1  foreach warp of 32 lanes in parallel do
2      lane_ix ← [1, ..., 32]
3      while more inputs do
4          t ← scan 32 tuples                          /* op_1 */
5          [...]                                  /* op_2 - op_{i-1} */
6          w ← number of work items
            after expansion in op_i
7          s ← data structure state op_i
8          t_buf, w_buf, s_buf ← t, w, s
9          while warp_any(w_buf > 0) do
10             a ← select_leader(w_buf)
11             t, w, s ← broadcast(a, t_buf, w_buf, s_buf)
12             for e ← lane_ix to w by 32 do
13                 process op_i expansion item e
14                 [...]                          /* op_{i+1} - op_n */
15             if lane_ix = a then
16                 w_buf ← 0
```

Figure 5: Pseudocode for a pipeline that applies Push-down Parallelism to $\mathtt{op}_i$. The strategy expands $\mathtt{op}_i$ with another level of parallelism.

Now, $\mathtt{op}_i$ is a hash probe that expands varying numbers of join matches per probe. Push-down parallelism performs the following steps. First, the number of join matches in each lane $w = [0, 4, 0, 6, 0, 28, 3, 0]$ is determined (line 6). Next, the state of each lane consisting of w, the tuple t, and data structure state s is written to local buffer variables w_{buf}, t_{buf}, and s_{buf} (lines 7 and 8). Then Push-down Parallelism enters a sequence of broadcast operations ① to ④ (lines 9 to 16) that finishes when no lane has remaining expansion items. For broadcast ①, Push-down Parallelism selects lane $a = 2$ with $w_{buf} = 4$ join matches as source •. The broadcast takes the values w_{buf}, t_{buf}, and s_{buf} and propagates them from lane 2 to the other lanes of the warp (line 11). Thus all warp lanes retrieve the probe-side tuple with it's current state. The build-side tuples are now retrieved from the hash bucket. The join matches ▭ are processed in a loop with adjacent hash buckets offsets for adjacent lanes (cf. lines 12 to 14). Push-down parallelism performs one loop iteration with the hash bucket offsets $e = [0, 1, 2, 3, \mathtt{x}, \mathtt{x}, \mathtt{x}, \mathtt{x}]$. During the iteration the subsequent operators $\mathtt{op}_{i+1}$ to $\mathtt{op}_n$ are executed (line 14). Now the hash probes from lane $a = 2$ are finished. This is marked by updating $w_{\text{buf}} = [0, 0, 0, 6, 0, 28, 3, 0]$ (lines 15 and 16). Push-down Parallelism continues with broadcast ②, which starts with the selection of lane $a = 4$ with 6 matches (line 10). The remaining broadcast procedure is unchanged and finishes by updating $w_{\text{buf}} = [0, 0, 0, 0, 0, 28, 3, 0]$ (lines 15 and 16). Broadcast ③ processes lane $a = 6$ with 28 matches. Here the larger number of matches necessitate 4 iterations of the loop in lines 12 to 14. The iterations process 8, 8, 8, and 4 matches. The broadcast finishes by updating $w_{\text{buf}} = [0, 0, 0, 0, 0, 0, 3, 0]$ and leaves 3 matches in lane $a = 7$ for broadcast ④. The join matches are processed, $w_{\text{buf}} = [0, 0, 0, 0, 0, 0, 0, 0]$ is updated, and the loop from line 9 exits. The pipeline starts over with fresh tuples.

Each broadcast takes the join matches from an individual lane and spreads them out across the warp. As consequence warps parallelize over the join matches instead of parallelizing over the scan table. This balances expansion processes *and* increases the memory efficiency for hash bucket reads. Push-down Parallelism yields preferable *coalesced memory access* patterns, which means that adjacent lanes access adjacent memory locations [15], whereas the standard approach uses slower sequential memory access.

3.3 Implementation

We implement Push-down Parallelism in DogQC by adding a code generation template to the query compiler. The implementation uses *warp primitives* via intrinsics, which allow lanes to exchange data and to perform collaborative computations [26]. E.g. `__sfhl_sync(..)` performs lane index-based data exchange and `__ballot_sync(..)` computes a predicate bitmask across a warp. We describe the implementation of lane buffering, leader selection, and broadcast operations with these intrinsics in the following.

Buffering Active Lanes. Lanes that receive work items during broadcast may already have an active tuple in register. To switch to a new work item, it is necessary to postpone processing of that tuple. This is done by buffering active tuples (line 8) before broadcast and leader selection. The buffer operation is local to each lane (i.e., lanes postpone only their own tuple). Consequently, buffering is as simple as writing each attribute value to a local buffer variable.

Leader Selection. During leader selection (line 10), Push-down Parallelism picks one lane as broadcast source and provides its lane index a to the other warp lanes. This is implemented with the following expression using only two warp intrinsics:

```
// select broadcast source lane
a = __ffs(__ballot_sync(w_buf>0,ALL));
```

The first primitive `__ballot_sync(...)` builds a bitmask of lanes that have remaining work items and shares it with all lanes. The second primitive `__ffs(...)` computes the index of the first 1-bit of the bitmask. The lane with index a is selected for broadcast.

Broadcast Operation. The broadcast operation (line 11) takes the buffered data from one lane a and distributes it to the other warp lanes. The following values are broadcasted: the attributes of the tuple $t_{\text{buf},a}$, the number of expansion items $w_{\text{buf},a}$, and the data structure state $s_{\text{buf},a}$, e.g., the hash bucket offset. The following code performs the broadcast for a tuple with two attributes and the hash bucket offset using *warp shuffle primitives*.

```
// gather w_buf, t_buf, and s_buf from lane a
w = __shfl_sync(w_buf,a);                          } w
o_orderdate = __shfl_sync(o_orderdate_buf,a);      }
o_orderkey = __shfl_sync(o_orderkey_buf,a);        } t
c_acctbal = __shfl_sync(c_acctbal_buf,a);          }
bucket_offs = __shfl_sync(bucket_offs_buf,a);      } s
```

The `__shfl_sync(...)` intrinsic takes the payload as first parameter and the source lane as second parameter. All lanes of the warp execute the instruction and obtain data from lane a. After the broadcast, each lane processes a distinct expansion work item (lines 12–14). E.g., hash bucket entries are obtained by adding the expansion index e to the base address of the hash bucket. In this way, warps consume the tuples from the hash bucket in coalesced iterations.

3.4 Planning for Push-down Parallelism

In DogQC, we select hash join operators for the application of Push-down Parallelism based on the build attributes. If the build is performed on other attributes than primary keys, hash buckets can contain multiple elements per key. We choose Push-down Parallelism to balance expansion processes during hash probes. For primary key build attributes, matching probes will retreive exactly one tuple. We choose the standard hash join as there is no expansion.

A fully-fledged system will include Push-down Parallelism in cost-based optimization as an alternative join operator. Similar to our current implementation, cost estimates can be based on build attribute statistics.

3.5 Usage Scenarios

Push-down Parallelism allows efficient execution of operators with expansion processes. The expansion may produce new tuples as the join in the previous example. Alternatively, expansions can be local and the operator passes on only one tuple, e.g., when processing the characters of string-typed attributes. For the latter case line 14 of the pseudocode in Figure 5 moves behind the **for**-loop.

By taking the parallelization level to the same level as the expansion process, Push-down Parallelism gives two main benefits. First, non-uniform distributions of the number of expansion items no longer cause expansion divergence. Second, memory accesses that are performed during expansion are transformed from sequential memory access to coalesced memory access. In the following, we discuss several scenarios for the application of Push-down Parallelism.

Joins. Joins between tables with varying key distributions are a poster child for the application of Push-down Parallelism. Existing GPU-based techniques restrict functionality by limiting the number of join matches, join conditions, or attributes stored in the hash table [17, 30, 32]. The restrictions limit divergence effects, but also lack support for important query plan options. DogQC handles varying key distributions, multi-predicate joins, and different payload sizes gracefully by using Push-down Parallelism to balance expansion work.

(Anti-) Semi Joins. Push-down Parallelism applies to (anti-) semi-joins with multiple match candidates (e.g., for combinations of equality and inequality predicates). The technique helps to balance the parallel evaluation of match candidates. However, the parallelization can prevent join strategies from early exit once the first match is found.

String Equality. Equals operations on string datatypes cause expansion divergence due to a varying numbers of characters in the strings. Push-down Parallelism expands the string characters across lanes and compares the characters in parallel. This reduces divergence effects from varying string lengths and increases memory efficiency by loading the string data using coalesced access.

Graph Processing. The node degree of real world graphs follows skewed distributions, e.g., power law [9]. Consequently, parallel graph algorithms are challenged by varying amounts of traversal work per node. Existing GPU techniques address these imbalances with node partitioning [21], edge partitioning [11], and compression [33]. Push-down parallelism naturally applies to the problem for relational graph representations.

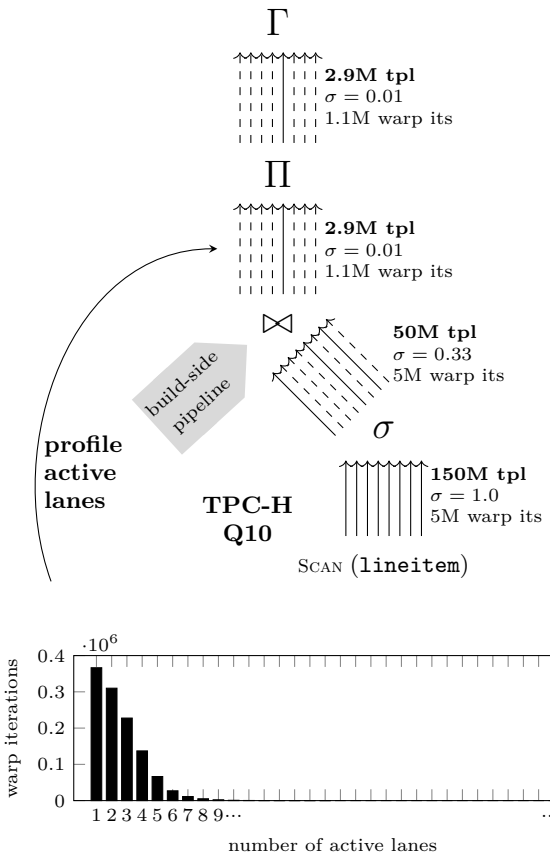

Figure 6: Analytic benchmark query with heavy filter divergence. After the filtering join operator most warp iterations have few active lanes.

4. FILTER DIVERGENCE

Filter divergence occurs in operators that inactivate some of the parallel lanes, for example filters and primary key-foreign key joins. The subsequent operations experience a lowered execution efficiency due to lane inactivity. This problem has been addressed by *stream compaction* [2] earlier; however, existing solutions are not suitable for compiled query pipelines because of their use of global synchronization barriers.

4.1 Poster Case 2

TPC-H Query 10 contains two selective operations on tuples from the `lineitem` table: a selection `l_returnflag = 'R'` and a sparse foreign key join with `l_orderkey = o_orderkey`. Figure 6 illustrates a pipeline that scans `lineitem` and then performs selection, join probe, projection, and aggregation. Compared to Section 3.1, the pipeline contains an additional projection for `l_extendedprice * (1-l_discount)`. The previous plan performed the projection in the build pipeline favoring a smaller hash table payload.

Again, we look at the number of *warp iterations* (cf. Section 3.1) in each pipeline stage to analyze the effect of the filters on execution efficiency. Starting with scan, the pipeline parallelizes 150 M `lineitem` tuples evenly across lanes. This requires $150\,\mathrm{M}/32 = 5\,\mathrm{M}$ warp iterations. The following filter with $\sigma = 0.33$ is likely to leave elements active in each warp. Consequently, the number of 5 M warp iterations remains con-

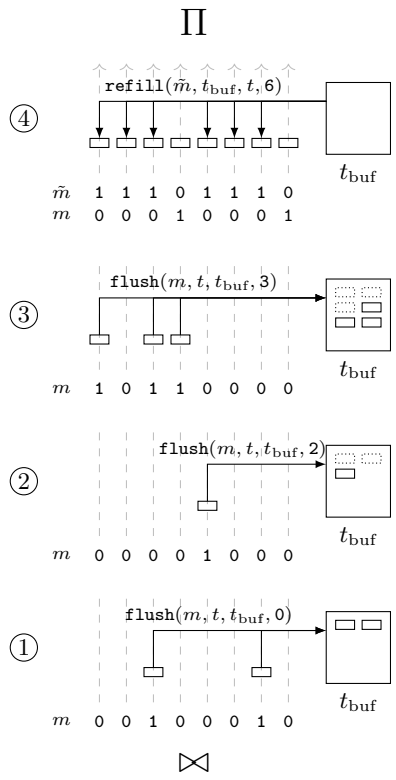

Figure 7: Illustration of Lane Refill that postpones processing of three low-activity iterations for full lane activity in the fourth iteration.

stant. Subsequently, the (single match) join probe produces 2.9 M tuples that are processed in 1.1 M warp iterations. Due to the selectivity of $\sigma = 0.01$ most lanes in the pipeline have become inactive and the remaining tuples are spread across warps. The histogram at the bottom of Figure 6 shows a profile of this pipeline stage, illustrating how many active lanes we measured in the 1.1 M executed warp iterations. Only few lanes are active in each warp causing a low execution efficiency that is carried through the subsequent projection and aggregation operators. Ideally, both operators would be processed with only $2.9\,\text{M}/32 = 90\,\text{K}$ warp iterations.

4.2 Lane Refill

Selective filters or sparse foreign key joins that trigger filter divergence situations are commonplace in analytic workloads [4]. The *Lane Refill* technique is a natural match to counter the imbalances caused by such operations. The technique we describe here resembles the mechanism proposed by Lang et al. [19] as *consume everything* strategy for SIMD processing. A similar idea was introduced by Polychroniou et al. [31] for a sequence of Bloom-filter bitmasks.

Lane Refill introduces *buffering operators* that control the lane activity during pipeline execution. The buffering operator is designed to work with a given *threshold*. If the lane activity drops below threshold there are two options:

1. There are insufficient buffered tuples. Active lanes are buffered and the pipeline starts over with fresh tuples.

2. There are sufficient buffered tuples to reach threshold and the tuples are reactivated in empty lanes.

```
LANE REFILL
1  foreach warp of 32 lanes in parallel do
2      n_buf ← 0
3      t_buf ← empty
4      while more inputs do
5          t ← scan 32 tuples                    /* op_1 */
6          [...]                                 /* op_2 - op_{i-1} */
7          m ← bitmask of active lanes
8          n_active ← popcount(m)
9          while n_buffer + n_active > T do
10             if n_active < T then
11                 n_buf ← refill(m̃, t, t_buf, n_buf)
12             execute op_i
13             [...]                             /* op_{i+1} - op_n */
14             m ← bitmask of active lanes
15             n_active ← popcount(m)
16         if n_active > 0 then
17             n_buf ← flush(m, t, t_buf, n_buf)
```

Figure 8: Pseudocode for a pipeline with Lane Refill between $\mathtt{op}_{i-1}$ and $\mathtt{op}_i$. The control flow proceeds with $\mathtt{op}_i$ only for lane activities above threshold T.

This strategy ensures that the operators succeeding the buffering operator always start with a lane activity above threshold. It is worth noting that one element buffer space for each lane is sufficient for any given threshold.

We show the pseudocode for the technique in Figure 8 and illustrate it in Figure 7. As an example, we assume a Lane Refill operator with threshold 7 (out of 8 lanes) that is placed after the sparse join of TPC-H Query 10. Figure 7 shows four iterations ① to ④ of the same warp receiving tuples from the sparse join. The boxes ▭ represent active lanes holding tuples. The first iteration receives two tuples from the join (pseudocode lines 1–6). Activity lies below threshold and the tuples are flushed to the buffer (lines 9 and 17). The pipeline starts over and the Lane Refill operator receives new tuples from the join. The following two iterations are flushed as well because the highest possible acitivity is 6 (out of 8) for three tuples from join plus three buffered tuples. In iteration ④, there are two fresh tuples and six buffered tuples. The empty lanes are refilled (lines 10–11) and the pipeline proceeds to the following operators with full lane activity. In the following, we show how Lane Refill is implemented in compiled query pipelines on GPUs.

4.3 Implementation

We implement Lane Refill in DogQC by introducing a buffering operator with the semantics shown in pseudocode Figure 8. The buffering operator is code generation-based, similar to the other operators in DogQC. The main challenges in adapting the approach by Lang et al. [19] are efficient GPU implementations for the balancing operations `flush` and `refill` and the application of warp parallelism.

The previous implementation of Push-down Parallelism performed lane communication via warp shuffles. This was possible because the only communication pattern used were gather operations. Lane Refill, however, uses scatter operations aswell. This is unsupported by warp shuffles, and therefore *shared memory* with communication via array-style indexing is better suited here.

Although shared memory and shuffle registers are both located on-chip, shared memory can perform slower when multiple lanes access the same memory bank [23]. However, further investigation of using warp shuffles only for the gather communication of lane refill showed no significant benefit over using solely shared memory.

Flush to Buffer. The `flush` operation is executed when the number of active lanes is below threshold and there are not enough buffer elements to restore sufficient activitiy. The remaining active lanes are written to empty buffer slots. `flush` takes a bitmask of active lanes m, the tuples t, the buffer t_{buf}, and the buffer count n_{buf} as input. Then `flush` computes the buffer destination `dest` that specifies the buffer position for each lane to write its active tuple to. This is done with the following code:

```
// warp prefix sum on active lanes
dest = __popc((m) & (pre_lanes)) + n_buf;
```

We look at an example with `8` lanes and lane activity $m =$ `[0,1,0,0,1,1,0,0]`. The bitmask `pre_lanes` marks all preceding lanes, e.g. lane `4` has `pre_lanes` = `[1,1,1,0,0,0,0,0]`. With the population count intrinsic `__popc(...)`, we count the set bits on preceding lanes. This gives us an exclusive prefix sum of the warp. With $n_{\text{buf}} = 2$ previously buffered elements, the destinations are `dest` = `[x,2,x,x,3,4,x,x]`.

Next, `flush` writes the tuples t from active lanes to the buffer t_{buf} at their respective destinations `dest`. This is done by scattering the tuple's attributes to shared memory, e.g.

```
// scatter to shared memory
l_extprice_buf[dest] = l_extprice;
o_orderdate_buf[dest] = o_orderdate;
```

Refill from Buffer. The `refill` operation is executed when the lane activity is below threshold *and* there are sufficient buffered tuples to reach threshold. The operation takes tuples from the buffer and reactivates them in passive lanes. `refill` receives the bitmask of passive lanes $\tilde{m}$, the tuples t, the buffer tuples t_{buf}, and the buffer count n_{buf} as input. To always maintain dense adjacent buffer elements, we push and pop the buffer content like a stack. To this end, we first compute the number of remaining buffer elements `n_remain` based on the buffer count and the number of empty lanes. Then we compute the buffer source index `src` with a warp prefix sum, similar to `flush`.

```
// warp prefix sum on passive lanes
src = __popc((inv_m) & (pre_lanes)) + n_remain;
```

After computing the buffer source index `src`, we can refill passive lanes from the buffer as shown below.

```
// gather from shared memory
if(src < n_buf) {
    l_extprice = l_extprice_buf[src];
    o_orderdate = o_orderdate_buf[src];
}
```

The code reads the attributes of buffered tuples from shared memory locations and stores them in registers by executing assignments to local variables. Note that we only load tuples from the buffer for the first n_{buf} passive lanes to account for the number of buffer elements.

4.4 Planning for Lane Refill

In the current version of DogQC, Lane Refill operators are placed manually into query plans. We insert balancing operators into pipelines with heavy filter divergence *if* there are multiple succeeding operators that can benefit from balanced processing. Section 5.4 specifies the queries where Lane Refill was applied in the context of the TPC-H benchmark.

In a fully-fledged system, optimizers will select insertion points for Lane Refill operators based on selectivity estimation. As the balancing operations can be executed with a low overhead, the negative impact of estimation errors is small. Optimizers that consider interesting orders, are affected by order changes due to balancing. Such optimizers, may leverage their ability to consider both plans with and without interesting orders during optimization.

4.5 Usage Scenarios

Lane Refill restores balanced lane activity in sequences of operators with filter divergence. The technique can be used after an operator that leaves execution in divergent stage (e.g. selection) before continuing with the next operator. Alternatively, Lane Refill can be used in succeeding iterations of the same operator (e.g. character comparisons in string equality) to restore lane activity between iterations. For the latter application, Lane Refill has the beneficial property to *preserve sequential order* of the iterations. This property is contrary to Push-down parallelism which parallelizes iterations. The sequential order can be leveraged by operators, such as regular expression matching with automata, where each iteration is dependent on the previous iterations. In the following we discuss several usage scenarios for Lane Refill.

Selection. Selection operators are a poster child for filter divergence. Database systems usually perform selection push-down to reduce workload sizes early. However, in data-parallel pipelines, the early selection does not reduce the workload size. Unless the full warp exits, lanes with filtered-out tuples still allocate the same processing resources. By filling the gaps with useful work, Lane Refill scales processing with the workload size.

Filter Join. Sparse foreign key joins occur in *both* normalized database workloads and in de-normalized star schema workloads. They recombine relations with only few matches for the join condition and filter out the majority of one of the relations. In the TPC-H benchmark join selectivities are usually around 0.1 and frequently go even lower [4]. Typically join filters go into effect during the join probe and leave lanes idle that have no match. By placing Lane Refill operators after a join probe, we can reactivate these idle lanes and allow them to perform more useful work.

String Pattern Matching. Database systems support string pattern matching with `LIKE`-predicates and regular expression (`regexp`) predicates. Most GPU-based systems, however, have very limited pattern matching capabilities, likely because of divergence effects [1, 7, 8, 12, 13]. Still there is existing work on GPU-based pattern matching. There is work on NFA-based `regexp` matchers [40], which parallelize over the states of the automaton. Albeit this parallelization strategy collides with per-tuple parallelization of GPU query engines. Other work on DFA-based matchers [35] uses per-string parallelism, which appears more suitable for query engines. During pattern matching, however, non-matching

strings reach rejecting states of the DFA early. Lane Refill can be used to reactivate those lanes with new tuples to make string pattern matching efficient. The property of Lane Refill to preserve sequential order is essential for following state transitions through DFAs.

Index Traversal. Index traversals are used to find tuples that match predicates. The hierarchical index structure is traversed from coarse-grained ranges to more fine-grained ranges to localize matching tuples. For regions with sparse population, traversal paths are often shorter than for densely populated regions. This leads to filter divergence during concurrent traversals. While B-Trees have relatively uniform path lengths, other index structures, e.g., for geospatial data [18], show more variation. To support such datatypes efficiently on GPUs, Lane Refill can be used to address these divergence effects during traversal.

5. EVALUATION

In this section, we evaluate the proposed techniques. We first evaluate the effect of Push-down Parallelism for expansion divergence. Then we evaluate the effect of applying Lane Refill to filter divergence. Next, we contrast Push-down Parallelism and Lane Refill when being applied to the same operation. Finally, we evaluate the overall performance of the divergence-optimized system against other state-of-the-art systems on CPU and GPU. In all experiments except for end-to-end performance (Figure 19) the execution times refer to GPU-resident data.

Query Processor. We evaluate the presented approach in the GPU-based query compiler DogQC[3]. DogQC follows an orthogonal approach to other GPU query processors. Instead of tuning *operator*-implementations for efficient GPU utilization, DogQC constructs pipelines from relatively simple operators and then applies tuning on the *pipeline level*. This two-step approach makes it more feasible to achieve both functionality *and* performance.

In the evaluation we use two versions of DogQC. The first version executes queries after the first step, which can cause heavy divergence during query processing. We call this version **DogQC naive**. The second version **DogQC opt** executes queries after the second step, which adds divergence balancing to increase processing efficiency on the GPU.

System. As experimentation platform, we use an NVidia RTX2080 GPU with 46 Streaming Multiprocessors (SMs) and 8 GB GPU Memory. We use Cuda 10.0 and `nvcc` V10.0.130 for JIT-compilation in all experiments but Figure 19, which uses `clang++9.0` to compile Cuda code. When not indicated differently, we use grid configurations of 80 warps per Streaming Multiprocessor (117,760 threads). This choice is due to sufficiently large grid sizes showing only small performance variations (cf. Figure 16). The GPU is placed in a workstation-class host system with 32 GB main-memory, operating an Intel Core i7-9800X CPU with Ubuntu 18.04 as operating system.

5.1 Effect of Push-down Parallelism

We first evaluate the benefit of Push-down Parallelism for expansion divergence. We execute a query that scans two relations and joins them with different join key distributions. We use a synthetic dataset where one relation has a dense primary key distribution and the other has one of the following key distributions:

pk-fk	Uniform distribution of foreign keys.
pk-8-fk	Each foreign key occurs 8 times.
pk-32-fk	Each foreign key occurs 32 times.
pk-zipf-fk	Foreign keys sampled from Zipfian distribution with $z = 0.75$ and $n = 10^7$.
pk-4zipf-fk	Foreign keys sampled from four Zipfian distributions with $z = 0.75$ and $n = 10^7$.

We generate join workloads for each of the distributions with 100 M build tuples and also 100 M result tuples. The first three workloads have an even number of 1 to 32 join matches per probe. Probes for pk-fk have exactly one match, probes for pk-8-fk have 8 matches, and 32 for pk-32-fk. With even match numbers we expect performance differences mainly due to the access method to hash buckets. The latter two workloads are non-uniform and the number of matches follows Zipfian distributions. The heaviest skew is for pk-zipf-fk with one probe matching the most frequent key 452 K times. For pk-4zipf-fk, the four frequent keys that occur 112 K times.

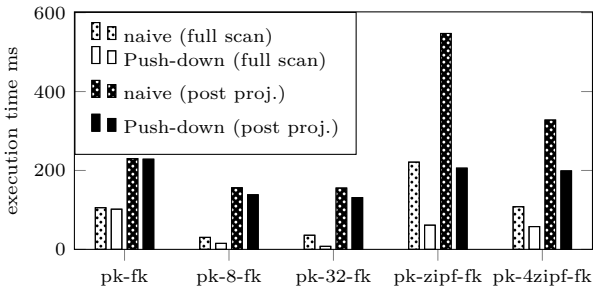

Figure 9: Divergence balancing for hash join with different build distributions. Push-down achieves robustness against skew and improves performance.

We show the results in Figure 9. The Figure reports execution times of the probe pipeline with the naive approach and with Push-down Parallelism for two different projection strategies. *Full scan* reads all attributes into registers during scan. *Post-proj* performs tuple-id based post projection.

We observe that Push-down Parallelism reduces execution times for all examined workloads by factors up to 4.2x. We discuss two effects that explain these improvements. The first effect is better load balance across threads, which becomes visible when comparing pk-zipf-fk to pk-4zipf-fk. The workloads have different levels of skew that affect the execution times of naive. Push-down parallelism achieves even execution times for both distributions.

The second effect is due to memory access patterns. Although the probes for pk-32-fk and pk-8-fk do not provoke load imbalance, the execution times improve. We attribute this to coalesced memory access, which means that adjacent lanes access adjacent memory locations in the hash buckets. This pattern is preferable on GPUs [15]. With Push-down Parallelism probes perform coalesced memory access with 8 or 32 lanes when executing pk-8-fk and pk-32-fk.

While we did not expect to observe an improvement for pk-fk, Push-down Parallelism reduces the execution times

[3]The source code of DogQC is available at
`https://github.com/henning1/dogqc`
`http://dbis.cs.tu-dortmund.de`

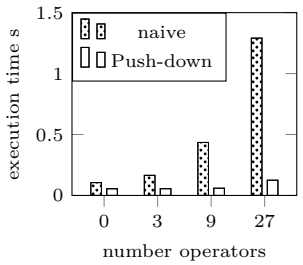

Figure 10: Varying numbers of operators after expansion.

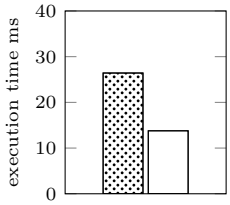

Figure 11: Poster Case 1 with Push-down parallelism.

by 4%. We explain this with an increased efficiency when handling hash collisions.

Looking at the two projection strategies, we observe that Push-down Parallelism provides benefits for both. Push-down parallelism improves by factors up to 2.7x for post-proj and by factors up to 4.2x for full scan. We attribute the higher benefit for full scan to the way Push-Down Parallelism channels tuple data to lanes with new join tuples. For post-proj only the tuple-id communicated via warp shuffles and other attributes are read from memory.

Varying Numbers of Operators. We evaluate the effect of varying the workload size that follows expansion divergence. This allows us to assess the impact of processing in divergent or in consolidated state. We append different numbers of projection operators to the pk-4zipf-fk workload and execute with the naive approach and with Push-down Parallism. We use configurations up to 27 operators to evaluate settings with high compute intensity. Figure 10 shows the experiment results.

Push-down parallelism improves throughput by factors that increase with the number of operators up to 10.3x. Further investigation showed that increasing the number of operators even further does not lead to higher factors. We attribute this effect to the compute load becoming the dominant part of the workload. The magnitude of the factor appears to be distribution dependent.

Poster Case 1. In Section 3.1, we discussed a query pipeline from TPC-H Query 10 with expansion divergence. Here we evaluate the effect of applying Push-down Parallelism in this pipeline to counter expansion divergence. We measure the execution time of the pipeline for a benchmark database with scale factor 25. We use a pipeline with the naive approach that has heavy expansion divergence in the join and we compare it to a pipeline that applies Push-down Parallelism in the join operator to counter expansion divergence. Figure 11 shows the experiment results. The naive approach has an execution time of 26.4 ms. Adding Push-Down Parallelism to the join operator of the pipeline reduces the execution time by a factor of 1.9x to 13.8 ms.

5.2 Effect of Lane Refill

We evaluate Lane Refill to counter filter divergence. The workload is a query that scans the TPC-H tables `lineitem` and `part`. The `lineitem` relation is filtered on `l_quantity` with *varying selectivities*, and then joined with `part` and aggregated. For Lane Refill, we place a balancing operator after the filter to restore lane activity. GPUs typically *oversubscribe* the number of warps to the number of streaming multiprocessors (SMs). This ability allows GPUs to hide

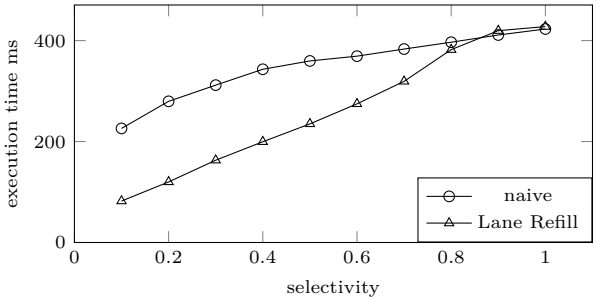

Figure 12: Effect of Lane Refill on filter divergence workload with one warp per SM. Execution times scale with the workload size when using Lane Refill.

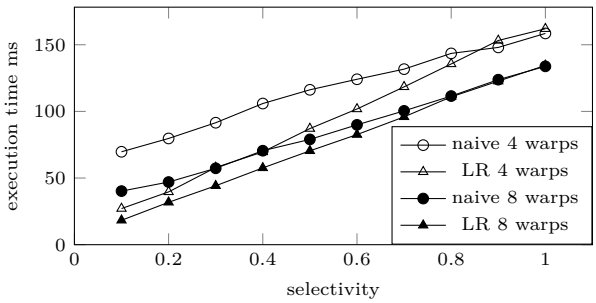

Figure 13: Effect of Lane Refill on filter divergence workload with multiple warps per SM. Lane Refill improves run-times for configurations with high degrees of warp-parallelism.

divergence effects to some extent. To understand the way Lane Refill works, we first suppress effects from oversubscription by using only *one warp per SM*. After that we perform another experiment with *multiple warps per SM*.

One Warp per SM. Figure 12 shows the results of the experiment with one warp per SM. If we set the filter to leave all tuples in the result (selectivity 1.0), we observe an execution time of 423 ms for the naive approach. The query becomes faster as we make the filter predicate more restrictive. For the naive strategy, that benefit is small, however: setting the selectivity to 0.4 improves performance by only 19% (343 ms). Only for very selective predicates, execution time noticeably drops, as shown in the graph for selectivity 0.1 (226 ms). This is because the naive approach can only benefit from filtering when *full warps* become inactive, but not if only subsets of the 32 lanes get filtered out.

Lane Refill, by contrast, benefits from restrictive predicates more directly and to a stronger extent. As we see in Figure 12, Lane Refill shows the desired linear scaling. For selectivity 0.1, execution time drops by 81% compared to a selectivity of 1.0. Compared to naive execution, this is a 2.8-fold improvement. Only for high selectivities (0.9 and 1.0) the balancing work introduces a small overhead up to 2%. We conclude that Lane Refill successfully prevents the GPU from working on inactive lanes and thus improves the processing efficiency.

Multiple Warps per SM. Figure 13 shows results for the same experiment, but we let the system overcommit and assign 4 and 8 warps to each SM. With 46 SMs on the RTX2080 GPU, this corresponds to 5,888 and 11,776 threads.

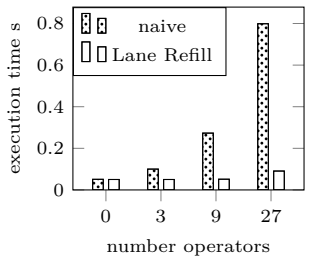

Figure 14: Varying numbers of operators that follow a filter.

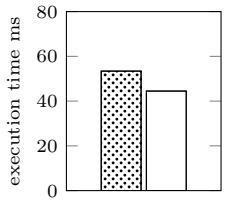

Figure 15: Applying Lane Refill in Poster Case 2.

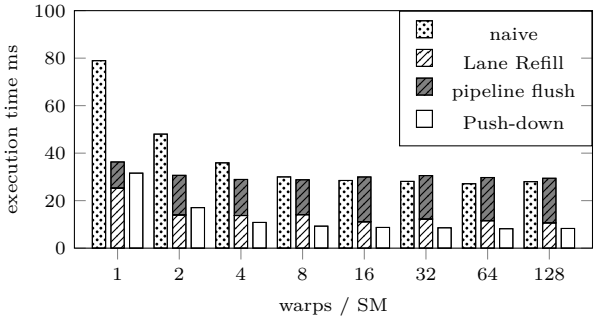

Figure 16: Push-down vs. Lane Refill when joining a Zipfian distribution. Push-down Parallelism is effective while Lane Refill suffers from pipeline flush.

As expected, overcommitting can hide some of the divergence effect that we saw in the previous experiment. Still, Lane Refill can better utilize the available resources, resulting in an performance advantage of 2.6x for the 4-warp configuration (70 ms vs. 27 ms) and 2.2x for the 8-warp configuration (40 ms vs. 18 ms). The balancing work causes a small overhead for high selecitivities up to 3.5%.

Varying Numbers of Operators. We evaluate the effect of varying the workload size that follows filter divergence. This allows us to assess the impact of processing in divergent or in consolidated state. We use a filter divergence workload and append low to high intensity compute loads by adding up to 27 projection operators. The workload scans 1.5 B tuples from the TPC-H table `lineitem` and filters on `l_quantity > 45` with selectivity 0.1. Figure 10 shows the experiment results.

For increasing numbers of operators the execution times of the naive approach go from 51 ms up to 799 ms. Lane Refill reduces the execution times to at most 91 ms (27 operators). We observe factors of improvement up to 8.8x, which corresponds to the lane utilization being raised from 0.1 (after the filter) close to 1.0 (after balancing). Further investigation of workloads with selectivity 0.2 support this explanation showing no better improvements than 5x. We suspected to observe a performance penalty caused by the lane refill computation for the data point with 0 operators. We attribute the absence of this to the high scan volume, which leaves compute resources available while servicing memory loads.

Poster Case 2. In Section 4.1 we presented a query pipeline from TPC-H Query 10 with filter divergence. Here we evaluate the effect of applying Lane Refill in this pipeline. We measure the execution time of the pipeline for a benchmark database with scale factor 25. We use the naive approach with filter divergence originating from the selection operator and from the sparse join operator. Then we compare the performance to a pipeline that adds a Lane Refill operator after the sparse join. Figure 15 shows the experiment results. The pipeline with the naive approach has an execution time of 53.4 ms. Adding the Lane Refill operator reduces the execution time of the pipeline by 1.2x to 44.5 ms.

5.3 Push-down Parallelism vs. Lane Refill

In this experiment, we apply Push-down Parallelism and Lane Refill to the same divergence problem. This allows us to determine whether each technique is best-suited for its respective divergence domain or if one technique may work for most cases. Expansion divergence can be viewed as filter divergence that occurs in steps of the same operation. E.g., when iterating through join matches, lanes with fewer expansion items act like filtered-out lanes in the current iteration. For the experiment, we use the workload **pk-zipf-fk** from Section 5.1, which joins a dense primary key with a Zipf-distributed foreign key. We use the naive approach, Lane Refill, and Push-down Parallelism for the join.

Observations. Figure 16 shows the results of the experiment. The figure shows execution times for different numbers of warps per Streaming Multiprocessor. The execution times for Lane Refill are split into regular work and pipeline flush. Pipeline flush represents work that is performed when all tuples are already scanned and only one remaining lane is active. We observe that Lane Refill can not improve over naive with regard to the best performing warp/SM configuration. For 1 warp/SM Lane Refill performs better than naive, but for larger warp/SM configurations, Lane Refill suffers from growing amounts of flush work. To achieve high performance, GPUs need many warps in flight. Therefore it is likely that heavy hitting tuples are isolated in warps. This prevents Lane Refill from performing effective balancing operations. Push-down Parallelism does not run into this problem because its balancing approach is effective, even when one tuple per warp is remaining. Push-down Parallelism improves over naive by 3.3x for the workload.

5.4 Overall Performance

This section evaluates the benefit of the proposed techniques when applied in an overall system. We analyze the impact on data imports, the additional resource demand, and the query performance for realistic workloads. We compare DogQC against two other systems: OmniSci [28], which uses GPU-based query compilation and MonetDB [5], which uses operator-at-a-time processing on CPUs. The experiments were performed with OmniSci 4.8.1 and MonetDB 11.33.3.

The analysis of query performance splits into two parts to address effects of divergence optimizations and end-to-end performance separately. The workloads for both parts are the TPC-H benchmark queries on a scale factor 25 GB database.

Import Cost of Dictionary Encoding. The approach used by DogQC works directly on variable length string data instead of building string dictionaries. This saves the cost of building string dictionaries during import. We quantify the cost with an experiment that uses OmniSci's parallel importer. We import a sequence of 100 M numeric values with 9 digits. One time the data is interpreted as `INTEGER` and another time the data is interpreted as `VARCHAR(10)`.

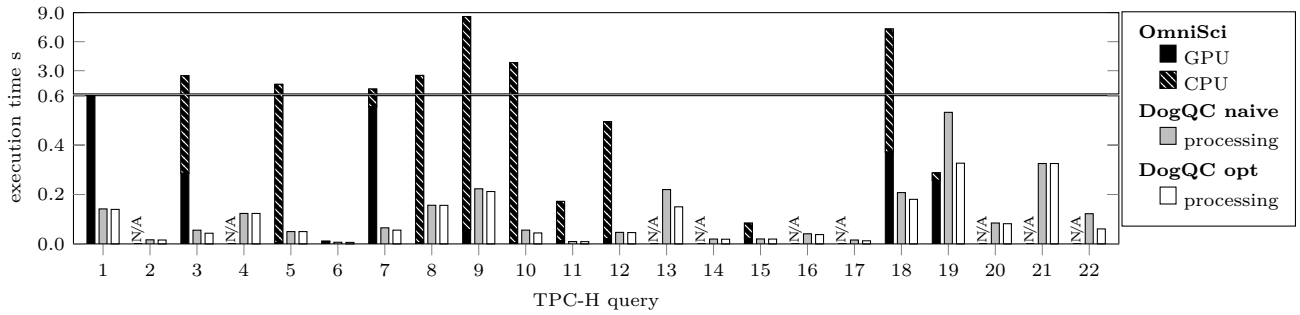

Figure 17: Execution times of DogQC for TPC-H benchmark queries (scale factor 25). The divergence optimizations improve query performance.

	Registers per thread	SMEM per block	Instruction footprint
Expansion naive	19	0 KB	2.36 KB
Expansion Push-down	29	0 KB	5.23 KB
Filter naive	36	0 KB	1.52 KB
Filter Lane Refill	33	1 KB	5.89 KB
Available capacity	255	64 KB	16 KB (L0)

Figure 18: Resource Consumption.

While the string-based import takes 16.16 s, the numeric import takes only 2.98 s. Importing the data with the use of a dictionary takes 5.4x longer. The shorter import times without dictionary encoding make a strong case for the processing approach of DogQC with variable length strings.

Resource Consumption. We analyze the additional demand for resources of the GPU cores when using divergence balancing. This allows us to assess whether the addition of divergence balancing to query pipelines causes bottlenecks during query execution. We profile the use of shared memory (SMEM), registers per thread, and the instruction footprint for the experiments from Sections 5.1 and 5.2. Figure 18 shows the results along with the available capacity.

Overall, we observe a low resource consumption. The highest relative demand with divergence balancing is 5.89 KB of the 16 KB L0-level instruction cache. The L0-level, however, is backed by three larger cache-levels [16]. We conclude that the balancing techniques have a low resource demand and there are sufficient open resources for complex queries.

Divergence Optimizations. We analyze the effect of applying divergence optimizations when processing realistic query workloads on the GPU. To this end, we analyze the execution times of the GPU systems DogQC and OmniSci. We use **DogQC naive** and for **DogQC opt** we add the following divergence optimizations: We replaced all join operator that do hash builds on non-primary key attributes with Push-down Parallelism join operators. Additionally we added eight Lane Refill balancing operators to the query plans; One to each of the Queries 4, 5, 7, 10, 15, 17, 19, and 20. We show the experiment results in Figure 17.

OmniSci was only able to execute 13 out of 22 queries. The execution times are split into GPU work and CPU work and range from 11 ms to 8599 ms. The highest time on GPU is 604 ms and 8542 ms on CPU. For nine of the supported queries, the CPU execution takes the majority of processing time. DogQC performs all processing on the GPU and was able to execute all TPC-H queries. **DogQC naive** has execution times between 7 ms and 532 ms and **DogQC opt** has execution times between 7 ms and 327 ms. Divergence optimizations reduced execution times by more than 5% for 10 out of 22 queries. The highest factors of improvement are 2.0x (Query 19) and 1.6x (Query 16). DogQC currently adds divergence balancing into query plans after optimization (cf. Sections 3.4 and 4.4). A future divergence-aware optimizer may find plans with a higher benefit.

In comparison of OmniSci and DogQC, we observe that OmniSci frequently falls back to slower CPU processing. This causes significantly higher execution times. DogQC's processing times were faster than OmniSci's by factors up to 68x for **DogQC naive** and by factors up to 86x **DogQC opt** for the divergence-optimized version. This shows that a fallback strategy for functionality that may be considered unsuitable for GPUs is disadvantageous. DogQC shows that it is preferable to include operations into compiled pipelines even when they cause heavy divergence. The highest benefit is achieved with additional divergence balancing.

End-to-End Performance. We evaluate the end-to-end performance of DogQC in comparison with MonetDB and OmniSci. OmniSci and DogQC are JIT compilation-based GPU systems. Therefore their end-to-end performance is affected by *two additional factors* that are *orthogonal* to divergence optimizations: The data transfer time between main memory and GPU, and the JIT-compilation time to generate machine code. Systems that keep the entire database in GPU memory only transfer the query results. DogQC is compatible with this mode of operation, but uses sequential input data transfers here. Asynchronous techniques that pipeline data transfers and processing [39, 12] can further reduce the transfer cost.

MonetDB runs on a two-socket server with Intel Xeon E5-2695 v2 CPUs and 256 GB main memory and the GPU hardware platform remains unchanged. For DogQC, we use the version **DogQC opt**.

The experiment results are shown in Figure 19. MonetDB's end-to-end execution times range from 142 ms up to 7464 ms. DogQC's end-to-end execution times range from 1188 ms to 3705 ms and were shorter than MonetDB's for 12 out of 22 queries. DogQC is faster only for longer running queries, where the lowered processing times outweigh the cost of data transfers and compilation. OmniSci was able to execute only 13 out of 22 queries. The end-to-end execution times range from 56 ms up to 8662 ms. The previous experiment showed that DogQC has lower processing times than OmniSci for all but one TPC-H query. End-to-end execution, however, is shorter for 8 of the 13 queries with OmniSci. The measurement shows that this effect is due to JIT compilation

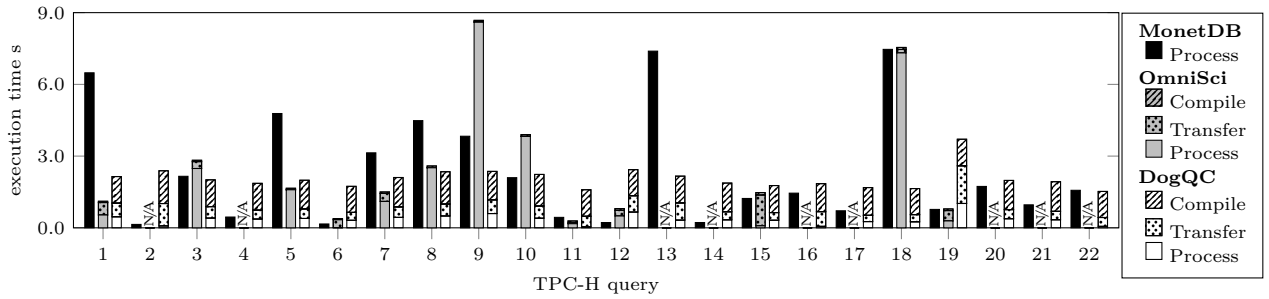

Figure 19: End-to-end performance for TPC-H benchmark queries with MonetDB (CPU) and OmniSci (GPU), and DogQC (GPU).

times. The current version of DogQC is not optimized for low compilation times and generates high-level Cuda code. Extending DogQC with a low-level code generator, such as LLVM, would reduce JIT-compilation times. The highest factors of improvement, that we observe in this experiment are 4.54x over MonetDB and 4.51x over OmniSci.

5.5 Usage Scenario: String Pattern Matching

In this paper, we proposed several usage scenarios for the presented divergence balancing techniques. To study their applicability, we exemplarily evaluate one of them. The workload is the test for a prefix of 50 characters in a dataset with titles of computer science articles. The dataset stems from DBLP[4] and was scaled up by 5x. Matching prefixes were scattered into random positions. We use two datasets with match rates of 1% and 32%, each has 21.5 M entries and an average title length of 76 characters. We process the workload with one warp per SM and each warp lane is responsible for one title at a time. We apply Lane Refill to reactivate lanes with rejected titles.

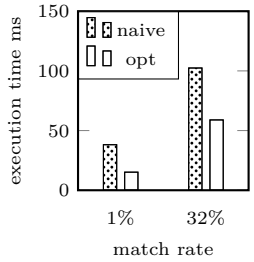

Figure 20: Balancing prefix tests.

The results of the experiment are shown in Figure 20. The workload with 1% matches has lower run-times than the workload with 32% matches. For lower match rates many strings are rejected early reducing the overall processing volume. We observe that divergence optimizations improve the performance of string prefix tests. For the 1% workload Lane Refill improves performance by 2.5x from 38 ms down to 15 ms. For the 32% workload the improvement is 1.7x from 102 ms down to 59 ms.

6. MORE RELATED WORK

In this section, we relate our approach to work that was not mentioned in one of the other sections. First we discuss work in the database context that uses the GPU feature *dynamic parallelism* to balance the use of parallel resources. Second we discuss other related GPU query processing techniques.

Dynamic Parallelism. Dynamic parallelism is a feature that allows GPUs to start new kernels from within a kernel [26]. The number of threads for the inner kernels can be chosen dynamically. Rui et al. [32] apply dynamic parallelism for sort-merge joins. Wang et al. [36] evaluate the feature for joins based on binary search and for regular expression matching. Liu et al. [22] propose the implementation of a MapReduce framework for GPUs with dynamic parallelism. Similar to Push-down Parallelism, dynamic parallelism adapts parallel resources to the characteristics of sub-problems. The main advantage of the approach is programmability. The downside, however, are costs for context switching. Chen et al. report overheads of up to 21x [10].

Pipelined GPU Query Processing. This work targets GPU query engines that implement pipelining via just-in-time compilation. In related work other means of pipelining have been proposed, such as in-cache processing [29] and kernel fusion [37]. Other related work that performs pipelining via just-in-time compilation [8, 38] may be susceptible to the presented divergence optimizations.

7. SUMMARY

In this research, we put the processing capabilities of data-parallel coprocessors for non-uniform database workloads to the test. DogQC introduces techniques, that allow us to gracefully align parallel processing units with work items, even when problems are heavily skewed. The evaluation analyzes different filter and join scenarios with distinct workload imbalances. We observe that the techniques Lane Refill and Push-down Parallelism are able to increase processing efficiency for these non-uniform workloads.

Existing query coprocessors typically avoid imbalances by working on a uniform surrogate (e.g. dictionary keys, materialization barriers). This has led to the perception, that GPUs have limited capabilities of processing irregular problems. DogQC conversely avoids the overhead of maintaining such additional data-structures and instead restores balance during non-uniform processing. This approach achieves a bigger functionality range and better performance than other query coprocessing engines. This is shown by support of the full set of TPC-H benchmark queries with best-in-class performance.

8. ACKNOWLEDGEMENTS

We would like to thank Florian Lüdiger for the experimental work on string pattern matching and the anonymous reviewers for their helpful comments and suggestions.

This work was supported by the DFG, Collaborative Research Center SFB 876, A2, and DFG Priority Program "Scalable Data Management for Future Hardware" (TE111/2-1).

[4] `https://dblp.org`

9. REFERENCES

[1] P. Bakkum and S. Chakradhar. Efficient data management for GPU databases. *High Performance Computing on Graphics Processing Units*, 2012.

[2] M. Billeter, O. Olsson, and U. Assarsson. Efficient stream compaction on wide SIMD many-core architectures. In *Proceedings of the conference on high performance graphics 2009*, pages 159–166. ACM, 2009.

[3] C. Binnig, S. Hildenbrand, and F. Färber. Dictionary-based order-preserving string compression for main memory column stores. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data*, pages 283–296. ACM, 2009.

[4] P. Boncz, T. Neumann, and O. Erling. TPC-H analyzed: Hidden messages and lessons learned from an influential benchmark. In *Technology Conference on Performance Evaluation and Benchmarking*, pages 61–76. Springer, 2013.

[5] P. A. Boncz, S. Manegold, M. L. Kersten, et al. Database architecture optimized for the new bottleneck: Memory access. In *VLDB*, pages 54–65, 1999.

[6] S. Borkar and A. A. Chien. The Future of Microprocessors. *Communications of the ACM*, 54(5):67–77, May 2011.

[7] S. Breß, H. Funke, and J. Teubner. Robust query processing in co-processor-accelerated databases. In *Proceedings of the 2016 International Conference on Management of Data*, pages 1891–1906. ACM, 2016.

[8] S. Breß, B. Köcher, H. Funke, S. Zeuch, T. Rabl, and V. Markl. Generating custom code for efficient query execution on heterogeneous processors. *VLDB Journal*, 27(6):797–822, 2018.

[9] A. Broder, R. Kumar, F. Maghoul, P. Raghavan, S. Rajagopalan, R. Stata, A. Tomkins, and J. Wiener. Graph structure in the web. *Computer networks*, 33(1-6):309–320, 2000.

[10] G. Chen and X. Shen. Free launch: optimizing GPU dynamic kernel launches through thread reuse. In *Proceedings of the 48th International Symposium on Microarchitecture*, pages 407–419. ACM, 2015.

[11] A. Davidson, S. Baxter, M. Garland, and J. D. Owens. Work-efficient parallel GPU methods for single-source shortest paths. In *2014 IEEE 28th International Parallel and Distributed Processing Symposium*, pages 349–359. IEEE, 2014.

[12] H. Funke, S. Breß, S. Noll, V. Markl, and J. Teubner. Pipelined Query Processing in Coprocessor Environments. In *Proceedings of the 2018 International Conference on Management of Data*, pages 1603–1618. ACM, 2018.

[13] M. Heimel, M. Saecker, H. Pirk, S. Manegold, and V. Markl. Hardware-oblivious parallelism for in-memory column-stores. *PVLDB*, 6(9):709–720, 2013.

[14] Intel Corporation. Accelerating x265 with Intel Advanced Vector Extensions 512 (Intel AVX-512). 2018.

[15] B. Jang, D. Schaa, P. Mistry, and D. Kaeli. Exploiting memory access patterns to improve memory performance in data-parallel architectures. *IEEE Transactions on Parallel and Distributed Systems*, 22(1):105–118, 2010.

[16] Z. Jia, M. Maggioni, J. Smith, and D. P. Scarpazza. Dissecting the NVidia Turing T4 GPU via Microbenchmarking. *arXiv preprint arXiv:1903.07486*, 2019.

[17] T. Kaldewey, G. Lohman, R. Mueller, and P. Volk. GPU join processing revisited. In *Proceedings of the Eighth International Workshop on Data Management on New Hardware*, pages 55–62. ACM, 2012.

[18] A. Kipf, H. Lang, V. Pandey, R. A. Persa, P. Boncz, T. Neumann, and A. Kemper. Approximate geospatial joins with precision guarantees. In *2018 IEEE 34th International Conference on Data Engineering (ICDE)*, pages 1360–1363. IEEE, 2018.

[19] H. Lang, A. Kipf, L. Passing, P. Boncz, T. Neumann, and A. Kemper. Make the most out of your SIMD investments: counter control flow divergence in compiled query pipelines. In *Proceedings of the 14th International Workshop on Data Management on New Hardware*, page 5. ACM, 2018.

[20] V. Leis, P. Boncz, A. Kemper, and T. Neumann. Morsel-driven parallelism: a NUMA-aware query evaluation framework for the many-core age. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, pages 743–754. ACM, 2014.

[21] H. Liu and H. H. Huang. Enterprise: breadth-first graph traversal on GPUs. In *SC'15: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–12. IEEE, 2015.

[22] L. Liu, Y. Zhang, M. Liu, C. Wang, and J. Wang. A-MapCG: an adaptive MapReduce framework for GPUs. In *2017 International Conference on Networking, Architecture, and Storage (NAS)*, pages 1–8. IEEE, 2017.

[23] X. Mei and X. Chu. Dissecting GPU memory hierarchy through microbenchmarking. *IEEE Transactions on Parallel and Distributed Systems*, 28(1):72–86, 2016.

[24] I. Müller, C. Ratsch, F. Faerber, et al. Adaptive String Dictionary Compression in In-Memory Column-Store Database Systems. In *EDBT*, pages 283–294, 2014.

[25] T. Neumann. Efficiently compiling efficient query plans for modern hardware. *PVLDB*, 4(9):539–550, 2011.

[26] NVidia Corporation. NVidia Kepler GPU Architecture. 2012.

[27] NVidia Corporation. NVidia Turing GPU Architecture. 2018.

[28] OmniSci Incorporated. OmniSciDB. `https://www.omnisci.com/`, 2019.

[29] J. Paul, J. He, and B. He. GPL: A GPU-based pipelined query processing engine. In *Proceedings of the 2016 International Conference on Management of Data*, pages 1935–1950. ACM, 2016.

[30] H. Pirk, S. Manegold, M. L. Kersten, et al. Accelerating Foreign-Key Joins using Asymmetric Memory Channels. In *ADMS@VLDB*, pages 27–35, 2011.

[31] O. Polychroniou and K. A. Ross. Vectorized Bloom filters for advanced SIMD processors. In *Proceedings of the Tenth International Workshop on Data Management on New Hardware*, page 6. ACM, 2014.

[32] R. Rui and Y.-C. Tu. Fast Equi-Join Algorithms on GPUs: Design and Implementation. In *Proceedings of*

the 29th International Conference on Scientific and Statistical Database Management, page 17. ACM, 2017.

[33] M. Sha, Y. Li, and K.-L. Tan. GPU-based Graph Traversal on Compressed Graphs. In *Proceedings of the 2019 International Conference on Management of Data*, pages 775–792. ACM, 2019.

[34] J. Sompolski, M. Zukowski, and P. Boncz. Vectorization vs. compilation in query execution. In *Proceedings of the Seventh International Workshop on Data Management on New Hardware*, pages 33–40. ACM, 2011.

[35] G. Vasiliadis, M. Polychronakis, S. Antonatos, E. P. Markatos, and S. Ioannidis. Regular expression matching on graphics hardware for intrusion detection. In *International Workshop on Recent Advances in Intrusion Detection*, pages 265–283. Springer, 2009.

[36] J. Wang and S. Yalamanchili. Characterization and analysis of dynamic parallelism in unstructured GPU applications. In *2014 IEEE International Symposium on Workload Characterization (IISWC)*, pages 51–60. IEEE, 2014.

[37] H. Wu, G. Diamos, S. Cadambi, and S. Yalamanchili. Kernel weaver: Automatically fusing database primitives for efficient GPU computation. In *2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 107–118. IEEE, 2012.

[38] H. Wu, G. Diamos, T. Sheard, M. Aref, S. Baxter, M. Garland, and S. Yalamanchili. Red fox: An execution environment for relational query processing on gpus. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, page 44. ACM, 2014.

[39] Y. Yuan, R. Lee, and X. Zhang. The Yin and Yang of processing data warehousing queries on GPU devices. *PVLDB*, 6(10):817–828, 2013.

[40] Y. Zu, M. Yang, Z. Xu, L. Wang, X. Tian, K. Peng, and Q. Dong. GPU-based NFA implementation for memory efficient high speed regular expression matching. In *ACM SIGPLAN Notices*, volume 47, pages 129–140. ACM, 2012.