Make the Most out of Your SIMD Investments: Counter Control Flow Divergence in Compiled Query Pipelines

Harald Lang
Technical University of Munich
Munich, Germany
harald.lang@in.tum.de

Andreas Kipf
Technical University of Munich
Munich, Germany
andreas.kipf@in.tum.de

Linnea Passing
Technical University of Munich
Munich, Germany
linnea.passing@in.tum.de

Peter Boncz
Centrum Wiskunde & Informatica
Amsterdam, The Netherlands
boncz@cwi.nl

Thomas Neumann
Technical University of Munich
Munich, Germany
thomas.neumann@in.tum.de

Alfons Kemper
Technical University of Munich
Munich, Germany
alfons.kemper@in.tum.de

ABSTRACT

Increasing single instruction multiple data (SIMD) capabilities in modern hardware allows for compiling efficient data-parallel query pipelines. This means GPU-alike challenges arise: *control flow divergence* causes underutilization of vector-processing units. In this paper, we present efficient algorithms for the AVX-512 architecture to address this issue. These algorithms allow for fine-grained assignment of new tuples to idle SIMD lanes. Furthermore, we present strategies for their integration with compiled query pipelines without introducing inefficient memory materializations. We evaluate our approach with a high-performance geospatial join query, which shows performance improvements of up to 35%.

ACM Reference Format:
Harald Lang, Andreas Kipf, Linnea Passing, Peter Boncz, Thomas Neumann, and Alfons Kemper. 2018. Make the Most out of Your SIMD Investments: Counter Control Flow Divergence in Compiled Query Pipelines. In *DaMoN'18: 14th International Workshop on Data Management on New Hardware, June 11, 2018, Houston, TX, USA.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3211922.3211928

1 INTRODUCTION

Integrating SIMD processing with database systems has been studied for more than a decade [21]. Several operations, such as selection [9, 16], join [1, 2, 7, 19], partitioning [14], sorting [4], CSV parsing [12], regular expression matching [18], and (de-)compression [10, 16, 20] have been accelerated using the SIMD capabilities of the x86 architectures. In more recent iterations of hardware evolution, SIMD instruction sets have become even more popular in the field of database systems. Wider registers, higher degrees of data-parallelism, and comprehensive support for integer data have increased the interest in SIMD and led to the development of many novel algorithms.

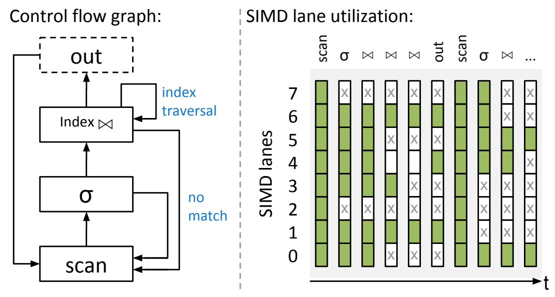


Figure 1: During query processing, individual SIMD lanes may (temporarily) become inactive due to different control flows. The resulting underutilization of vector-processing units causes performance degradations. We propose efficient algorithms and strategies to fill these gaps.

SIMD is mostly used in interpreting database systems that use the *column-at-a-time* or *vector-at-a-time* [3] execution model. Compiling database systems like HyPer [6] barely use it due to their data-centric *tuple-at-a-time* execution model [13]. In such systems, therefore, SIMD is primarily used in scan operators [9] and in string processing [12].

With the increasing vector-processing capabilities for database workloads in modern hardware, especially with the advent of the AVX-512 instruction set, query compilers can now vectorize entire query execution pipelines and benefit from the high degree of data-parallelism [5]. With AVX-512, the width of vector registers increased to 512 bit, allowing for processing of an entire cache line in a single instruction. Depending on the bit-width of the attribute values, up to 64 elements can be packed into a single register.

Vectorizing entire query pipelines raises new challenges. One such challenge is keeping all SIMD lanes busy during query evaluation. Efficient algorithms are required to counter underutilization of vector-processing units (VPUs). In [11], this issue was addressed by introducing (memory) materialization points immediately after each vectorized operator. However, with respect to the more strict definition of pipeline breakers given in [13], materialization points can be considered pipeline breakers because tuples are evicted from registers to slower (cache) memory. In this work, we present algorithms and strategies that do not break pipelines. Further, our approach can be applied at intra-operator level and not only at operator boundaries.

The remainder of this paper is organized as follows. In Section 2, we describe the potential performance degradation caused by underutilization in holistically vectorized pipelines. In Section 3, we present efficient algorithms to counter underutilization, and in Section 4, we present strategies for integrating these algorithms with compiled query pipelines. Experimental evaluation of the proposed algorithm is given in Section 5, and our conclusions are given in Section 6.

2 VECTORIZED PIPELINES

The major difference between a scalar (i.e., non-vectorized) pipeline and a vectorized pipeline is that in the latter, multiple tuples are pushed through the pipeline at once. This impacts *control flow* within the query pipeline. In a scalar pipeline, whenever the control flow reaches any operator, it is guaranteed that there is *exactly one* tuple to process (tuple-at-a-time). By contrast, in a vectorized pipeline, there are several tuples to process. However, because the control flow is not necessarily the same for all tuples, some SIMD lanes may become inactive when a conditional branch is taken. Such a branch is only taken if *at least one* element satisfies the branch condition. This implies that a vector of length n may contain up to $n-1$ *inactive* elements, as depicted in Figure 1.

Inactive elements typically result from *filter* operations. Tuples that do not satisfy some given predicate are disqualified by setting the corresponding SIMD lane as inactive, but the actual non-qualifying values remain in the vector register. Nevertheless, the control flow must enter the subsequent operator if at least one element in the vector register qualifies. Thus, disqualified elements cause underutilization in the subsequent operator(s).

Another source of inactive lanes are *search operations*, for example, in pointer-based data structures such as trees and hash tables, which typically occur in joins. Because some search operations may be completed earlier than others, some SIMD lanes may become (temporarily) inactive. This is an inherent problem when traversing irregular pointer-based data structures in a SIMD fashion [17].

Generally, all conditional branches within the query pipeline are potential sources of control flow *divergence* and, therefore, a source of underutilization of VPUs. To avoid underutilization through divergence we need to dynamically assign new tuples to idle SIMD lanes, possibly at multiple "points of divergence" within the query pipeline. We refer to this process as pipeline *refill*.

3 REFILL ALGORITHMS

In this section, we present our refill algorithms for AVX-512[1], which we later integrate into compiled query pipelines (cf., Section 4). These algorithms essentially copy new elements to *desired positions* in a destination register. In this context, these desired positions are the lanes that contain inactive elements. The active lanes are identified by a small bitmask (or simply *mask*), where the i^{th} bit corresponds to the i^{th} SIMD lane. A SIMD lane is active if the corresponding bit is set, and vice versa. Thus, the bitwise complement of the given mask refers to the inactive lanes and, therefore, to the write positions of new elements. We distinguish between two cases

[1]We refer the reader to Appendix A.1 which briefly describes the key features of the AVX-512 instruction set architecture (ISA) used in our algorithms.

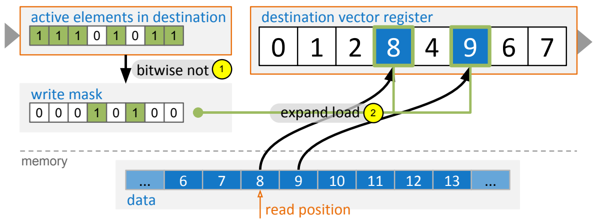


Figure 2: Refilling empty SIMD lanes from memory using the AVX-512 expand load instruction.

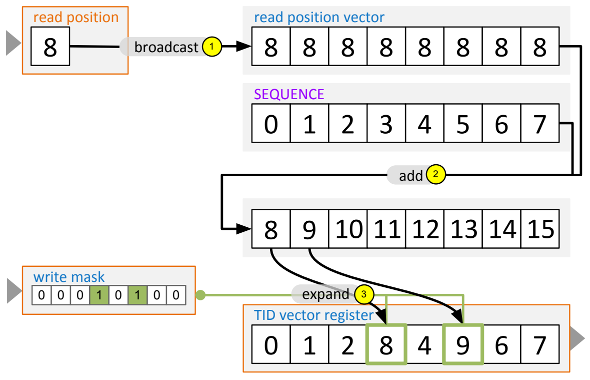


Figure 3: In general, the associated tuple identifiers (TIDs) of loaded values need to be carried along the query pipeline. The TIDs are derived from the current read position and assigned to a TID vector register.

as follows: (i) where new elements are copied from a source memory address and (ii) where elements are already in vector registers.

3.1 Memory to Register

Refilling from memory typically occurs in the table scan operator, where contiguous elements are loaded from memory (assuming a columnar storage layout). AVX-512 offers the convenient `expand load` instruction that loads contiguous values from memory directly into the desired SIMD lanes (cf., Figure 2). One mask instruction (`bitwise not`) is required to determine the *write mask* and one vector instruction (`expand load`) to execute the actual load. Overall, the simple case of refilling from memory is supported by AVX-512 directly out of the box.

Nevertheless, the table scan operator typically produces an additional output vector containing the tuple identifiers (TIDs) of the newly loaded attribute values. The TIDs are derived from the current read position and are used, for example, to (lazily) load attribute values of a different column later on or to reconstruct the tuple order (cf., Figure 3).

3.2 Register to Register

Moving data between vector registers is more involved. In the most general case we have a source and a destination register that contain both active and inactive elements at *random* positions. The goal is to move as many elements as possible from the source to the

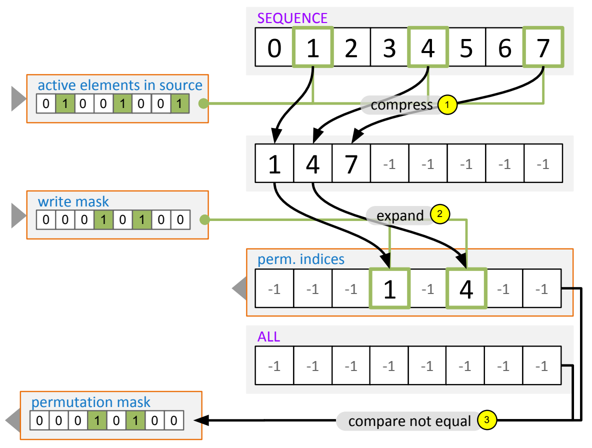


Figure 4: Computation of the permutation indices and mask based on positions of the active elements in the source register and the inactive elements in the destination register.

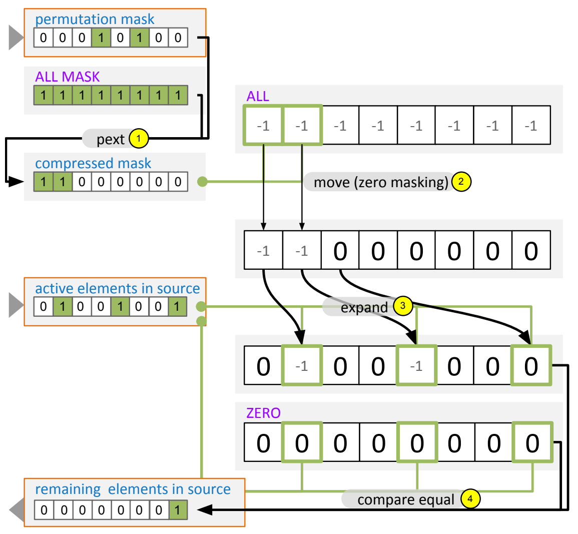


Figure 5: If not all elements could be moved from the source to the destination register, the source mask needs to be updated accordingly.

destination. This can be achieved using a single masked `permute` instruction. However, first, the *permutation indices* must be computed based on the positions of active elements in the source and the destination vector registers. This is illustrated in Figure 4, where, as in the previous examples, the *write mask* refers to the inactive lanes in the destination register. In total, three vector instructions are required to compute the permutation indices and an additional permutation mask. The latter is required in case the number of active elements in the source is smaller than the number of empty lanes in the destination vector.

Once the permutation indices are computed, elements can be moved between registers accordingly. Notably, the algorithm can be adapted to move elements directly instead of computing the permutation indices first. However, if elements need to be moved between more than one source/destination vector pair, the additional cost of computing the permutation amortizes immediately with the second pair. In practice, the permutation is typically applied multiple times, for example, when multiple attributes are pushed through the pipeline or to keep track of the TIDs.

In the general case there are no guarantees about the number of (active) elements nor their positions within the vector register. For example, the elements in the source may not be entirely consumed or the destination vector may still contain inactive elements. Thus it is necessary to update source and destination masks accordingly. Updating the destination mask is straightforward by using a `bitwise or` with the previously computed permutation mask. Updating the source mask is less obvious as illustrated in Figure 5. As the figure shows, updating the source mask is as expensive as computing the permutation. However, if it is guaranteed that all source elements fit into the destination vector, this phase of the algorithm can be skipped altogether.

Depending on the position of the elements, cheaper algorithms can be used. Especially when the vectors are in a *compressed state*, meaning that the active elements are stored contiguously, it is considerably cheaper to prepare the permutation (compare Listing 2 and 3).

These two foundational SIMD algorithms cover the extreme cases, that is, where (i) active elements are stored at random positions and (ii) active elements are stored contiguously. Based on these cases, the algorithms can be easily adapted such that only one vector needs to be compressed, which is useful when vector registers are used as tiny buffers because those should always be in a compressed state to achieve the best performance. In total, there are four different algorithms. Each algorithm has two different flavors: (i) where all elements from the source register are guaranteed to fit into the destination register or (ii) where not all elements can be moved and therefore elements remain in the source register. Owing to space constraints, we do not show all variants here, but have released the source code[2] under the BSD license.

4 REFILL STRATEGIES

We discuss the integration of these refill algorithms in data-centric *compiled* query pipelines. Such pipelines turn a query operator pipeline into a for-loop, and the code generated by the various operators is nested bottom-up in the body of such a loop [13]. Relational operators in this model generate code in two methods, namely, `consume()` and `produce()`, which are called in a depth-first traversal of the query tree: `produce()` code is generated before generating the code for the children, and `consume()` afterwards.

The main idea of data-centric execution with SIMD is to insert checks for each operator that control the number of tuples in play, i.e. if-statements nesting the rest of the body. Such an if-statement makes sure its body only gets executed if the SIMD registers are sufficiently full. Generally speaking, operator code processes input SIMD data computed by the outer operator and *refills* the registers it works with and the ones it outputs.

We identify two *base strategies* for applying this refilling.

[2]Source code: https://github.com/harald-lang/simd_divergence

Listing 1: Code skeleton of a buffering operator.

```
[...]
auto active_lane_cnt = popcount(mask);
if (active_lane_cnt + buffer_cnt < THRESHOLD
    && !flush_pipeline) {
 [...] // Buffer the input.
}
else {
 const auto bail_out_threshold = flush_pipeline ? 0
                                                : THRESHOLD;
 while (active_lane_cnt + buffer_cnt > bail_out_threshold) {
  if (active_lane_cnt < THRESHOLD) {
   [...] // Refill lanes with buffered elements.
  }
  //===----------------------------------===//
  // The actual operator code and
  // consume code of subsequent operators.
  [...]
  //===----------------------------------===//
  active_lane_cnt = popcount(mask);
 }
 if (likely(active_lane_cnt != 0)) {
  [...] // Buffer the remaining elements.
 }
}
mask = 0; // All lanes empty (Consume Everything semantics).
[...]
```

4.1 Consume Everything

The Consume Everything strategy allocates additional vector registers that are used to *buffer* tuples. In the case of underutilization, the operator *defers* processing of these tuples. This means the body will not be executed in this iteration (if-condition not satisfied) but instead (else) the active tuples will be moved to these buffer registers. It uses the refill algorithms from the previous section, both to move data to the buffer and to emit buffered tuples into the unused lanes in a subsequent iteration. Listing 1 shows the code skeleton as it would be generated by a *buffering* operator. The important thing to note here, is that all SIMD lanes are empty when the control flow returns to the previous operator.

Compared to a scalar pipeline, this strategy only requires a minor change to the push model: handling a special case when the pipeline execution is about to terminate, flushing the buffer(s). The essence is that buffering only takes place in SIMD registers and it specifically does not cause extra in-memory materialization, which would break the operator pipeline.

4.2 Partial Consume

As the name suggests, the second base strategy no longer expects the `consume()` code to process the entire input. The consume code can decide to defer execution by returning the control flow to the previous operator and leave the active elements in the vector registers. Naturally, these register lanes must not be overridden or modified by other operators in the next iteration. We refer to these elements (or to their corresponding lanes) as being *protected*. Another way of looking at a protected lane is that the lane is *owned* by a different operator. When an *owning* operator completes processing a tuple, it transfers the ownership to the subsequent operator. Alternatively, if the tuple is disqualified, it gives up ownership to allow a tuple producing operator to assign a new tuple to the corresponding lane.

Lane protection requires additional bookkeeping on a per operator basis. Each operator must be able to distinguish between tuples that (i) have just arrived, (ii) have been protected by the operator in an earlier iteration and (iii) tuples that have already advanced to later stages in the pipeline.

4.3 Discussion and Implications

The two strategies are not mutually exclusive. Within a single pipeline, both strategies can be applied to individual operators so long as buffering operators are aware of protected lanes (*mixed* strategy). Moreover, the query compiler might decide to *not* apply any refill strategy to certain operators. Especially, when a sequence of operators is quite cheap, divergence might be acceptable so long as the costs for refill operations are not amortized. Naturally, this is a physical query optimization problem, we leave for future work. Nevertheless, we briefly discuss the advantages and disadvantages, as this is the first work in which we present the basic principles of vector-processing in compiled query pipelines.

As mentioned above, *Consume Everything* requires additional registers which increases the register pressure and may lead to spilling. *Partial Consume* allocates additional registers as well, but these are restricted to (smaller) mask registers. Therefore, it is unlikely to be affected by (potential) performance degradation due to spilling.

The second major difference lies in the costs of refilling empty lanes. In a pipeline that follows the Partial Consume strategy, the very first operator, that is, the pipeline source, is responsible for refilling empty lanes. If other operators experience underutilization, they return the control flow to the previous operator while retaining ownership of the active lanes. This cascades downward until the source operator is reached. All operators between the source and the operator that returned the control flow may be subject to underutilization because all lanes in later stages are protected. The costs of refilling, therefore, depend on the length of the pipeline and the costs of the preceding operators. In general, the costs increase in the later stages. Nevertheless, Partial Consume can improve query performance if it is applied only to the very first operators. By contrast, the refilling costs of buffering operators do not depend on the pipeline length. Instead, the crucial factor governing these costs is the number of required buffer registers. The greater the number of buffers, the greater the number of `permute` instructions that need to be executed. Whereas the number of required buffers depends on (i) the number of attributes passed along the pipeline and optionally on (ii) the number of registers required to save the internal state of the operator (e.g., a pointer to the current tree node).

5 EVALUATION

We evaluated our foundational algorithmic principle for SIMD lane optimization in a modern database application scenario: geospatial data processing. Specifically, we selected a high-performance point-polygon join [8]. The geospatial join is more computationally demanding than a traditional hash join and is therefore a candidate for being SIMD optimized. It further can be parameterized with

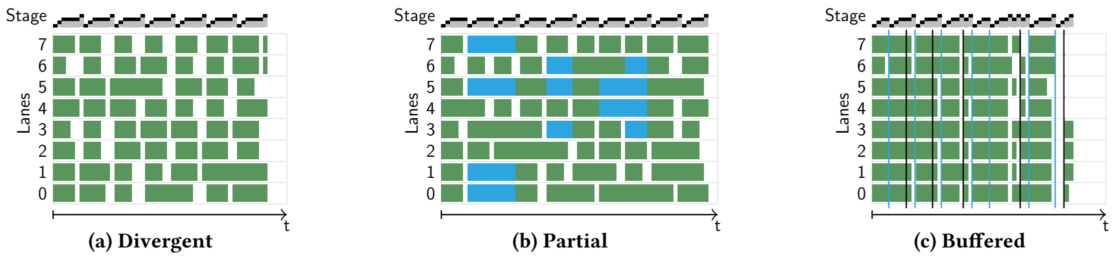


Figure 6: SIMD lane utilization when joining 64 points with the NYC neighborhoods polygons. – In (a) no refilling is performed to visualize the divergence. – (b) shows a Partial Consume throughout the entire pipeline with the minimum required utilization set to 50%. Lanes colored in blue are protected. – (c) uses the Consume Everything strategy which performs refills when the utilization falls below 75%. The blue lines indicate a write to buffer registers, black lines a read.

respect to precision, which influences divergence, thus allowing us to evaluate our algorithms with different settings.

In this section we first briefly introduce the geospatial join and its characteristics with respect to control flow divergence, as well as the workloads/datasets used in our experiments.

5.1 Approximate Geospatial Join

Our approximate geospatial join [8] uses a quadtree-based hierarchical grid to approximate polygons. The grid cells are stored in a specialized radix tree, where the cell size corresponds to the level within the tree structure (larger cells are stored closer to the root node and vice versa). During join processing we perform (prefix) lookups on the radix tree. Each lookup is separated into two stages: First, we check for a *common prefix* of the query point and the indexed cells. The common prefix allows for fast *filtering* of query points. If the query point does not share the common prefix, there are no join partners. The actual tree traversal takes place in the second stage. We traverse the tree starting at the root node until we hit a leaf node (which contains a reference to the matching polygon).

An important property of our approximate geospatial join operator is that it can be configured to guarantee a certain precision. In the experiments, we used 60-, 15-, and 4-meter precision (as in [8]). The higher the precision guarantee, the smaller are the cells at the polygon boundaries, which in turn increases the total number of cells and, more importantly, the height of the radix tree. In general, the probability of control flow divergence during index lookups increases with the tree height. Throughout our experiments the tree height is ≤ 6.

5.2 Workloads

In our experiments, we join the boroughs, neighborhoods, and census blocks polygons of New York City (NYC) with randomly generated points, uniformly distributed within the bounding box of the corresponding polygonal dataset. The datasets vary in terms of the total number of polygons and complexity (w.r.t. the number of vertices). A common characteristic of the three polygon sets is that most of the polygons within each set have roughly the same size in terms of covered area. Therefore, to demonstrate the effectiveness of our algorithms, we add a fourth polygon set "Manhattan" with higher "divergence potential" where we mix neighborhoods and census blocks. Specifically, we use smaller polygons in Manhattan.

Table 1: Polygon datasets

	number of polygons	avg. number of vertices
boroughs	5	662.2
neighborhoods	289	29.6
census	39184	12.5
manhattan	594	7.7

Table 1 summarizes the relevant metrics of the polygon datasets, and Table 3 summarizes the metrics of the corresponding radix tree, including the probability distribution of the number of search steps during the tree traversal.

5.3 Query Pipeline

The query pipeline of our experiments (point-polygon join) consists of four stages:

(1) Scan point data (source)
(2) Prefix check
(3) Tree traversal
(4) Output point-polygon pairs (sink)

Stages (2) and (3) are subject to control flow divergence, with (3) being significantly costlier than (2). For simplicity the produced output (point-polygon pairs) is not further processed. We compile the pipeline in four different flavors:

Divergent: Refers to the baseline pipeline without divergence handling, thus the pipeline follows Consume Everything semantics. The code of subsequent operators is executed if at least one lane is active.

Partial: Partial Consume is applied to stages (2) and (3), which also affects the scan operator because it needs to be aware of protected lanes.

Buffered: Follows Consume Everything semantics with register buffers in stage (3). We check the lane utilization after each traversal step. The operator state is buffered whenever the lane utilization falls below a certain threshold. For our evaluation, we require at least 6 out of 8 SIMD lanes to be active. Divergence in stage (2) is not handled at all.

Table 2: Hardware platforms

	Intel Knights Landing	Intel Skylake-X
model	Phi 7210	i9-7900X
cores (SMT)	64 (x4)	10 (x2)
SIMD [bit]	2×512	2×512
max. clock rate [GHz]	1.3	4.5
L1 cache	64 KiB	32 KiB
L2 cache	1 MiB	1 MiB
L3 cache	-	14 MiB

Mixed: Applies Partial Consume until stage (2) and buffering in stage (3). As described in Section 4.3, lane protection inherently causes underutilization of SIMD lanes, but it may perform better than buffering when installed closer to the pipeline source.

Figure 6 illustrates the lane utilization for the different (base) strategies employed herein. The indicator on top of each plot shows the control flow within the query pipeline.

5.4 Results

All experiments are conducted on the Knights Landing (KNL) platform using 128 threads and on the Skylake-X (SKX) using 10 threads. Table 4 shows the performance baseline of our geospatial join without divergence handling in units of million tuples per second. Figure 7 shows the relative performance of the `Partial`, `Buffered`, and `Mixed` pipelines compared to the baseline.

As expected, application of Partial Consume to the entire pipeline, exacerbates the divergence issue (cf., Section 4.3 and Figure 6b), resulting in a 43% performance degradation on KNL and approximately 30% on SKX in the worst case. By contrast, the `Buffered` and `Mixed` pipelines show no or just minor performance degradation for slightly divergent workloads (up to 7% in worst case). With the least divergent workload, boroughs with 60 m precision, refilling barely amortizes. Most searches in the radix tree terminate within the first two search steps which makes the second stage quite cheap, mitigating the performance impact of control flow divergence.

On KNL, we observe a correlation between the relative performance speedups and the precision parameter of the geospatial join. For all workloads, the `Buffered` pipeline shows the highest performance improvements on KNL (up to 21%). By contrast on SKX, the `Mixed` pipeline exhibits better performance in 7 out of 12 experiments. The performance degradation of the `Mixed` pipeline over `Buffered` on KNL can be explained with its limited out-of-order execution capabilities. A pipeline refill introduces several branches and leads to a more complex control flow. Multiple refills within a single pipeline (i.e., stage 2 and 3) exacerbate this issue. Moreover, a partial consume issues (mostly) scalar instructions, which is unfavorable for a SIMD-focused architecture like KNL.

The `Buffered` and `Mixed` pipelines achieve their best performance under the Manhattan workload, which has the highest divergence of all workloads, showing the high performance gain (up to 35% in this case) that our approach can generate.

In summary, our results show that divergence handling can increase query performance for divergent workloads substantially, while only introducing minimal overhead for non-divergent pipelines.

6 CONCLUSIONS

We presented efficient refill algorithms for vector registers by using the latest SIMD instruction set, AVX-512. In contrast to previous work, we do not rely on predefined lookup tables nor do our algorithms introduce costly memory materializations, which makes them suitable for data-centric query compilation. We identified and presented two basic strategies for applying refilling to compiled query pipelines for preventing underutilization of VPUs. We evaluated our approach by applying to a high-performance geospatial join query that involved traversing an irregular pointer-based data structure (a radix tree), showing that our strategies can efficiently handle control flow divergence, while yielding up to 35% higher throughput.

7 ACKNOWLEDGEMENTS

This work has been partially supported by the German Federal Ministry of Education and Research (BMBF) grant `01IS12057` (FASTDATA and MIRIN), and the DFG projects `NE1677/1-2` and `KE401/22`. This work is further part of the TUM Living Lab Connected Mobility (TUM LLCM) project and has been funded by the Bavarian Ministry of Economic Affairs, Energy and Technology (StMWi) through the Center Digitisation.Bavaria, an initiative of the Bavarian State Government.

REFERENCES

[1] Cagri Balkesen, Gustavo Alonso, Jens Teubner, and M. Tamer Özsu. 2013. Multi-Core, Main-Memory Joins: Sort vs. Hash Revisited. *PVLDB* 7, 1 (2013), 85–96. http://www.vldb.org/pvldb/vol7/p85-balkesen.pdf

[2] Cagri Balkesen, Jens Teubner, Gustavo Alonso, and M. Tamer Özsu. 2013. Main-memory hash joins on multi-core CPUs: Tuning to the underlying hardware. In *29th IEEE International Conference on Data Engineering, ICDE 2013, Brisbane, Australia, April 8-12, 2013.* 362–373. https://doi.org/10.1109/ICDE.2013.6544839

[3] Peter A. Boncz, Marcin Zukowski, and Niels Nes. 2005. MonetDB/X100: Hyper-Pipelining Query Execution. In *CIDR 2005, Second Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 4-7, 2005, Online Proceedings.* 225–237. http://cidrdb.org/cidr2005/papers/P19.pdf

[4] Jatin Chhugani, Anthony D. Nguyen, Victor W. Lee, William Macy, Mostafa Hagog, Yen-Kuang Chen, Akram Baransi, Sanjeev Kumar, and Pradeep Dubey. 2008. Efficient implementation of sorting on multi-core SIMD CPU architecture. *PVLDB* 1, 2 (2008), 1313–1324. http://www.vldb.org/pvldb/1/1454171.pdf

[5] Tim Gubner and Peter Boncz. 2017. Exploring Query Compilation Strategies for JIT, Vectorization and SIMD. In *Eighth International Workshop on Accelerating Analytics and Data Management Systems Using Modern Processor and Storage Architectures, ADMS 2017, Munich, Germany, September 1, 2017.*

[6] Alfons Kemper and Thomas Neumann. 2011. HyPer: A hybrid OLTP&OLAP main memory database system based on virtual memory snapshots. In *Proceedings of the 27th International Conference on Data Engineering, ICDE 2011, April 11-16, 2011, Hannover, Germany.* 195–206. https://doi.org/10.1109/ICDE.2011.5767867

[7] Changkyu Kim, Eric Sedlar, Jatin Chhugani, Tim Kaldewey, Anthony D. Nguyen, Andrea Di Blas, Victor W. Lee, Nadathur Satish, and Pradeep Dubey. 2009. Sort vs. Hash Revisited: Fast Join Implementation on Modern Multi-Core CPUs. *PVLDB* 2, 2 (2009), 1378–1389. http://www.vldb.org/pvldb/2/vldb09-257.pdf

[8] Andreas Kipf, Harald Lang, Varun Pandey, Raul Alexandru Persa, Peter Boncz, Thomas Neumann, and Alfons Kemper. 2018. Approximate Geospatial Joins with Precision Guarantees. In *34rd IEEE International Conference on Data Engineering, ICDE 2018, Paris, France, April 16-19, 2018.*

[9] Harald Lang, Tobias Mühlbauer, Florian Funke, Peter A. Boncz, Thomas Neumann, and Alfons Kemper. 2016. Data Blocks: Hybrid OLTP and OLAP on Compressed Storage using both Vectorization and Compilation. In *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016.* 311–326. https://doi.org/10.1145/2882903.2882925

Table 3: Metrics of radix tree

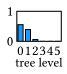
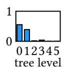
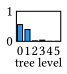
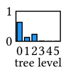
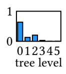
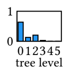
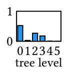
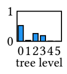
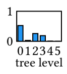
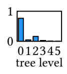
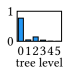
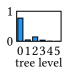
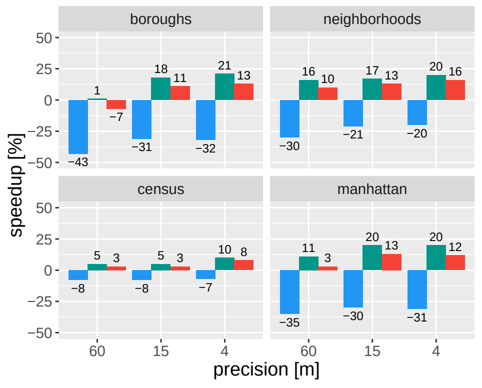
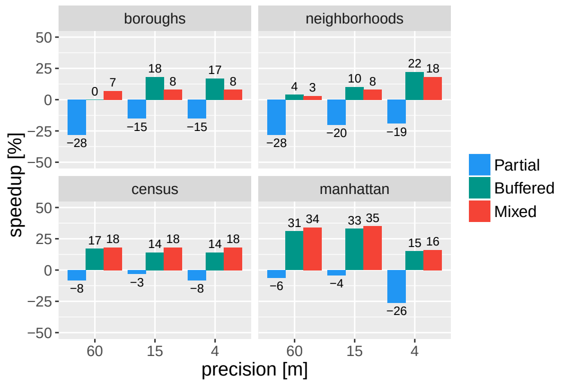
polygons	boroughs			neighborhoods			census			manhattan		
precision [m]	60	15	4	60	15	4	60	15	4	60	15	4
# of cells [M]	0.08	1.27	20.7	0.11	0.79	13.2	6.08	6.52	34.6	0.14	0.19	1.36
tree size [MiB]	1.39	168	168	25.3	139	139	1162	1205	1205	21.1	29.3	29.3
tree traversal depth												

Table 4: Performance baseline [Mtps] (divergent pipeline)

polygons	boroughs			neighborhoods			census			manhattan		
precision [m]	60	15	4	60	15	4	60	15	4	60	15	4
Knights Landing	5128	3399	3582	3414	2275	2304	856	814	955	4134	3559	3606
Skylake-X	1620	905	908	1002	762	689	382	373	372	1048	910	1049



(a) Knights Landing



(b) Skylake-X

Figure 7: Performance speedups

Listing 2: Generic refill algorithm

```
struct fill_rr {
 __mmask8 permutation_mask; __m512i permutation_idxs;
 // Prepare the permutation.
 fill_rr(__mmask8 src_mask, __mmask8 dst_mask) {
  const __m512i src_idxs =
      _mm512_mask_compress_epi64(ALL, src_mask, SEQUENCE);
  const __mmask8 write_mask = _mm512_knot(dst_mask);
  permutation_idxs =
    _mm512_mask_expand_epi64(ALL, write_mask, src_idxs);
  permutation_mask = _mm512_mask_cmpneq_epu64_mask(
    write_mask, permutation_idxs, ALL);
 }
 // Move elements from 'src' to 'dst'.
 void apply(const __m512i src, __m512i& dst) const {
  dst = _mm512_mask_permutexvar_epi64(
          dst, permutation_mask, permutation_idxs, src);
 }
 void update_src_mask(__mmask8& src_mask) const {
  __mmask8 comp_mask = _pext_u32(~0u, permutation_mask);
  __m512i a = _mm512_maskz_mov_epi64(comp_mask, ALL);
  __m512i b = _mm512_maskz_expand_epi64(src_mask, a);
  src_mask =
    _mm512_mask_cmpeq_epu64_mask(src_mask, b, ZERO);
 }
 void update_dst_mask(__mmask8& dst_mask) const {
  dst_mask = _mm512_kor(dst_mask, permutation_mask);
 }
};
```

Listing 3: Refill algorithm for compressed vectors

```
struct fill_cc {
 __mmask8 permutation_mask; __m512i permutation_idxs;
 uint32_t cnt;
 // Prepare the permutation.
 fill_cc(const uint32_t src_cnt, const uint32_t dst_cnt) {
  const auto src_empty_cnt = LANE_CNT - src_cnt;
  const auto dst_empty_cnt = LANE_CNT - dst_cnt;
  // Determine the number of elements to be moved.
  cnt = std::min(src_cnt, dst_empty_cnt);
  auto d = (dst_empty_cnt >= src_cnt) ? dst_cnt
                                      : src_empty_cnt;
  const __m512i d_vec = _mm512_set1_epi64(d);
  // Note: No compress/expand instructions required.
  permutation_idxs = _mm512_sub_epi64(SEQUENCE, d_vec);
  permutation_mask = ((1u << cnt) - 1) << dst_cnt;
 }
 // Move elements from 'src' to 'dst'.
 void apply(const __m512i src, __m512i& dst) const {
  dst = _mm512_mask_permutexvar_epi64(
          dst, permutation_mask, permutation_idxs, src);
 }
 void update_src_cnt(uint32_t& src_cnt) const {
  src_cnt -= cnt;
 }
 void update_dst_cnt(uint32_t& dst_cnt) const {
  dst_cnt += cnt;
 }
};
```

[10] Daniel Lemire and Christoph Rupp. 2017. Upscaledb: Efficient Integer-Key Compression in a Key-Value Store using SIMD Instructions. *Inf. Syst.* 66 (2017), 13–23. https://doi.org/10.1016/j.is.2017.01.002

[11] Prashanth Menon, Andrew Pavlo, and Todd C. Mowry. 2017. Relaxed Operator Fusion for In-Memory Databases: Making Compilation, Vectorization, and Prefetching Work Together At Last. *PVLDB* 11, 1 (2017), 1–13. http://www.vldb.org/pvldb/vol11/p1-menon.pdf

[12] Tobias Mühlbauer, Wolf Rödiger, Robert Seilbeck, Angelika Reiser, Alfons Kemper, and Thomas Neumann. 2013. Instant Loading for Main Memory Databases. *PVLDB* 6, 14 (2013), 1702–1713. http://www.vldb.org/pvldb/vol6/p1702-muehlbauer.pdf

[13] Thomas Neumann. 2011. Efficiently Compiling Efficient Query Plans for Modern Hardware. *PVLDB* 4, 9 (2011), 539–550. http://www.vldb.org/pvldb/vol4/p539-neumann.pdf

[14] Orestis Polychroniou, Arun Raghavan, and Kenneth A. Ross. 2015. Rethinking SIMD Vectorization for In-Memory Databases. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015.* 1493–1508. https://doi.org/10.1145/2723372.2747645

[15] Orestis Polychroniou and Kenneth A. Ross. 2014. Vectorized Bloom filters for advanced SIMD processors. In *Tenth International Workshop on Data Management on New Hardware, DaMoN 2014, Snowbird, UT, USA, June 23, 2014.* 6:1–6:6. https://doi.org/10.1145/2619228.2619234

[16] Orestis Polychroniou and Kenneth A. Ross. 2015. Efficient Lightweight Compression Alongside Fast Scans. In *Proceedings of the 11th International Workshop on Data Management on New Hardware, DaMoN 2015, Melbourne, VIC, Australia, May 31 - June 04, 2015.* 9:1–9:6. https://doi.org/10.1145/2771937.2771943

[17] Bin Ren, Gagan Agrawal, James R. Larus, Todd Mytkowicz, Tomi Poutanen, and Wolfram Schulte. 2013. SIMD parallelization of applications that traverse irregular data structures. In *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization, CGO 2013, Shenzhen, China, February 23-27, 2013.* 20:1–20:10. https://doi.org/10.1109/CGO.2013.6494989

[18] Evangelia A. Sitaridi, Orestis Polychroniou, and Kenneth A. Ross. 2016. SIMD-accelerated regular expression matching. In *Proceedings of the 12th International Workshop on Data Management on New Hardware, DaMoN 2016, San Francisco, CA, USA, June 27, 2016.* 8:1–8:7. https://doi.org/10.1145/2933349.2933357

[19] Jens Teubner and René Müller. 2011. How soccer players would do stream joins. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2011, Athens, Greece, June 12-16, 2011.* 625–636. https://doi.org/10.1145/1989323.1989389

[20] Wayne Xin Zhao, Xudong Zhang, Daniel Lemire, Dongdong Shan, Jian-Yun Nie, Hongfei Yan, and Ji-Rong Wen. 2015. A General SIMD-Based Approach to Accelerating Compression Algorithms. *ACM Trans. Inf. Syst.* 33, 3 (2015), 15:1–15:28. https://doi.org/10.1145/2735629

[21] Jingren Zhou and Kenneth A. Ross. 2002. Implementing database operations using SIMD instructions. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data, Madison, Wisconsin, June 3-6, 2002.* 145–156.

A APPENDIX

A.1 AVX-512 Instruction Set

In this section we briefly describe the key features of the AVX-512 instruction set that we use in our algorithms in Section 3.

Mask instructions: Almost all AVX-512 instructions support *predication.* These instructions allow to perform a vector operation only on those vector components (or lanes) specified by a given bitmask, where the i^{th} bit in the bitmask corresponds to the i^{th} lane. For example an `add` instruction in its simplest form requires two (vector) operands and a destination register that receives the result. In AVX-512 the `add` instruction exists in two additional flavors:

(1) **Merge masking:** The instruction takes two additional arguments, a *mask* and a source register, for example, `dst = mask_add(src,mask,a,b)`. The addition is performed on the vector components in `a` and `b` specified by the `mask`. The remaining elements, where the mask bits are `0`, are copied from `src` to `dst` at their corresponding positions.

(2) **Zero masking:** The functionality is basically the same as that of merge masking, but instead of specifying an additional source vector, all elements in `dst` are set to zero if the corresponding bit in the mask is not set. Zero masking is therefore (logically) equivalent to merge masking with `src` set to zero: `maskz_add(mask,a,b)` $\equiv$ `mask_add(`$\vec{0}$`,mask,a,b)`. Thus, zero masking is a special case of merge masking.

Masked instructions can be used to prevent individual vector components from being altered, e.g., `x = mask_add(x,mask,a,b)`.

Typically, masks are created using comparison instructions and stored in special mask registers, which is a significant improvement over earlier SIMD instruction sets, in which these masks were stored in vector registers.

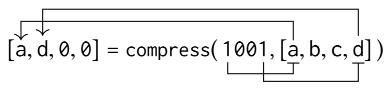
Permute: The `permute` instruction shuffles elements within a vector register according to a given index vector:

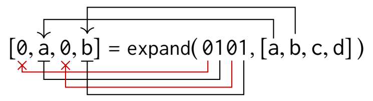
$$\underbrace{[\texttt{d,a,d,b}]}_{\text{result vector}} = \texttt{permute(}\underbrace{[\texttt{3,0,3,1}]}_{\text{index vector}}\texttt{,}\ \underbrace{[\texttt{a,b,c,d}]}_{\text{input vector}}\texttt{)}.$$

Our register-to-register refill algorithms are designed around this instruction (cf., the `apply()` functions in Listings 2 and 3).

It is noteworthy, that the `permute` instruction has already been available in earlier instruction sets. But due to the doubled register size, twice as many elements can now be processed at once. Further, in our application, we achieve a four times higher throughput compared to AVX2. The reason is, that a refill is basically a *merge* operation of the content of two vector registers. In combination with merge masking, refilling can be performed using a single instruction, whereas with AVX2 two instructions need to be issued (`permute` + `blend`).

Compress / Expand: A refill requires cross-lane operations which are inefficient in SSE/AVX architectures. Dynamically permuting elements in SIMD registers used to be a tedious task that often induced significant overheads, such as additional accesses into predefined lookup tables [9, 11, 15]. The key instructions introduced with AVX-512, to efficiently solve these type of problems, are called `compress` and `expand`. `Compress` stores the active elements (indicated by a bitmask) contiguously into a target register, and `expand` stores the contiguous elements of an input at certain positions (specified by a *write mask*) in a target register:





Both instructions come in two flavors: (i) read/write from/to memory and (ii) directly operate on registers.

Details about the AVX-512 instruction set architecture can be found in the Intel Intrinsics Guide [3].

[3]https://software.intel.com/sites/landingpage/IntrinsicsGuide/