Rethinking MIMD-SIMD Interplay for Analytical Query Processing in In-Memory Database Engines

Lennart Schmidt
TU Dresden
Dresden, Germany
lennart.schmidt1@tu-dresden.de

Johannes Pietrzyk
TU Dresden
Dresden, Germany
johannes.pietrzyk@tu-dresden.de

Juliana Hildebrandt
TU Dresden
Dresden, Germany
juliana.hildebrandt@tu-dresden.de

Alexander Krause
TU Dresden
Dresden, Germany
alexander.krause@tu-dresden.de

Dirk Habich
TU Dresden
Dresden, Germany
dirk.habich@tu-dresden.de

Wolfgang Lehner
TU Dresden
Dresden, Germany
wolfgang.lehner@tu-dresden.de

ABSTRACT

Exploiting parallelism is the name of the game for executing analytical queries with low latency in in-memory database engines. Most prominently, modern general-purpose CPUs, which continue to dominate the area of computing units, offer high-computational power through two data-oriented parallel paradigms: MIMD and SIMD. Unfortunately, as both parallel paradigms exhibit different programming models and memory access patterns, exploiting both in a combined fashion is challenging. However, recent hardware advances for SIMD on CPUs have relaxed the restrictions on SIMD-friendly memory access patterns. The availability and performance of alternative access patterns has improved significantly compared to the state-of-the-art of a pure linear access pattern. As we will demonstrate in this paper, these advancements pave the way for a unified parallelization approach that harnesses both MIMD and SIMD in a combined fashion, offering a novel and promising way for an efficient analytical query processing.

1 INTRODUCTION

Processing of complex analytical queries with low latency and high throughput on large amounts of data is a major challenge in our data-driven world [31]. To master this challenge, database engines constantly adapt to novel hardware features on a technical level [3, 4, 7, 9, 15, 18, 21, 32]. In the last two decades, we have seen numerous hardware advances, in particular with respect to main memory, computing units, and networks [6, 25, 29]. For example, the capacity of main memory has dramatically increased allowing to keep the full (transactional) database in main memory [10]. Also, novel variants such as non-volatile or high-bandwidth main memory have been developed with an impact, e.g., on data structure design [21], operator implementation [24], or recovery mechanisms [2]. In the area of computing units, the core count increased and internal techniques like advanced instruction set extensions, pre-fetching, or branch-prediction improved within modern general-purpose CPUs [1, 12, 19]. Furthermore, alternative computing units like GPUs, FPGAs, or any combination of them provide a wide field of opportunities to adapt database engines [7, 14, 15, 23]. Relevant advances have also taken place in the area of networks so that modern networks are larger, more efficient, and offer more services [17, 28].

Even with the continuous evolution of computing units, the enduring dominance of general-purpose CPUs in this field remains a significant fact. In particular, modern general-purpose CPUs offer high-computational power through three different sources of parallelism [26]: *thread-level parallelism (based on the Multiple Instruction Multiple Data – MIMD – parallel paradigm)*, *data-level parallelism (based on the Single Instruction Multiple Data – SIMD – parallel paradigm)*, and *instruction-level parallelism*. State-of-the-art analytical database engines leverage all three sources of parallelism in a more or less common way to reduce query latency:

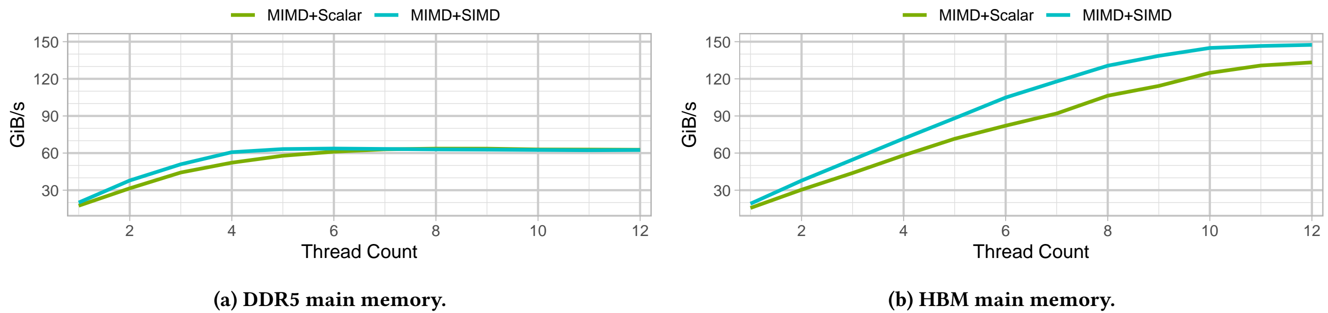
Thread-level parallelism: is usually applied on the level of individual query operators or pipelines (intra-operator/intra-pipeline parallelism) by distributing the input data equally among threads (physical cores) [16, 22]. That means the same operator/pipeline is simultaneously executed on multiple (logically) disjoint data partitions.

Data-level parallelism: is achieved by explicitly using SIMD instructions within query operators/pipelines. A single SIMD instruction processes multiple data elements simultaneously, increasing the single-thread performance. The SIMD instructions are usually applied to contiguous data elements in main memory [12, 26].

Instruction-level parallelism: is achieved by applying the same operation to a vector of elements [5] or by compiling operations into intertwined pipeline machine code [20].

While instruction-level parallelism is about executing several instructions in sequence on loaded or cached data, thread-level and data-level parallelism are used to process data in parallel. Moreover, thread-level and data-level parallelism complement each other and should usually be combined to exploit the full potential of modern CPUs. Nevertheless, thread-level often has a higher priority than data-level parallelism since the main memory bandwidth is already fully saturated with the use of thread-level parallelism. This aspect is clearly shown in the diagram in Figure 1a illustrating the measured memory throughput for calculating an aggregating sum of a large array of integer values with increasing thread-level parallelism (increasing number of physical cores) as well as with

This paper is published under the Creative Commons Attribution 4.0 International (CC-BY 4.0) license. Authors reserve their rights to disseminate the work on their personal and corporate Web sites with the appropriate attribution, provided that you attribute the original work to the authors and CIDR 2025. 15th Annual Conference on Innovative Data Systems Research (CIDR '25). January 19-22, Amsterdam, The Netherlands



(a) DDR5 main memory. **(b) HBM main memory.**

Figure 1: Comparing single-threaded throughput for aggregating sum with data in different main memory technologies.

(+SIMD) and without (+Scalar) using data-level parallelism. As depicted, the use of data-level parallelism only reduces the number of required threads to achieve the maximum throughput of roughly 60GiB/s. The hardware foundation for these experiments was a recent Intel Xeon Max 9648 (Sapphire Rapids architecture) with 48 physical cores on a socket. This hardware is also the basis for all further experimental results in the remainder of the paper. Moreover, the socket was divided into four logical NUMA nodes using Sub-NUMA clustering-4 (SNC4). Every resulting NUMA region with 64GB DDR5 (regular DRAM) and 16GB HBM2 memory (high-bandwidth memory) is associated with 12 physical cores. Unless explicitly stated otherwise, we use Intel AVX512 as the SIMD instruction set extension for our SIMD experiments.

Since our hardware foundation does not only have regular DRAM, we repeated the same experiment with the whole data in HBM2 as well. Interestingly, when executing the same operation on data in HBM (cf. Figure 1b), exploiting data-level parallelism becomes mandatory to utilize the interconnect fully. This experimental result clearly shows that the joint utilization of MIMD and SIMD is gaining in importance with developments such as of HBM. However, due to diverging granularities of parallelism and scopes of data accesses, the state-of-the-art interplay of MIMD and SIMD execution paradigms requires different approaches, which creates algorithmic overhead when combining them. Therefore, we argue that it is time to fundamentally rethink the MIMD-SIMD interplay on modern general-purpose CPUs through a unified memory access approach.

Our Contribution and Outline

On the thread-level scope, data is divided into a set of coarse-grained logical partitions, which are processed by individual threads. On the data-level scope, the coarse-grained partitions are further subdivided into fine-grained logical partitions, which are processed by individual SIMD register lanes. This paper proposes a novel concept to coalesce thread-level and data-level parallelism for efficient query processing in in-memory database engines. With our approach it is possible to exploit both levels of parallelism with the same access patterns and thus alleviates the combination of both. Additionally, our unified approach offers a number of advantages, e.g., the performance is similar to state-of-the-art and even outperforms it in some cases. To support our claims, we make the following contributions in this paper:

- In Section 2, we revisit the state-of-the-art for thread-level and data-level parallelism for executing analytical queries with low latency in in-memory database engines in more detail.
- Based on these preliminary remarks, Section 3 presents a novel, data-partitioned SIMD processing concept by introducing a suitable data access pattern. This novel approach transfers the established thread-level data-partitioned concept to SIMD.
- In the following Sections 4 and 5, we apply the unified parallelization strategy for the processing of data structured according to columnar and n-ary storage model and show the respective advantages by comparing them with the state-of-the-art.

Finally, we close the paper with a brief summary and an outlook on future research in Section 6.

2 PRELIMINARIES

As already mentioned, parallelism is the name of the game for an efficient processing of analytical queries on general-purpose CPUs. According to Flynn's classification [11], modern CPUs offer the following hardware parallelization opportunities: (i) **Multiple Instruction Multiple Data (MIMD)** and (ii) **Single Instruction Multiple Data (SIMD)**. In the following, we will discuss the state-of-the-art approaches for both. As a representative running analytical example, we compute the aggregating sum over a large column or array of integer data (contiguous memory area). A scalar implementation of this aggregating sum sequentially iterates over the array and adds up the values one after the other.

2.1 MIMD

CPUs offering MIMD have a number of processing elements (PE) – cores – that operate asynchronously and independently. That means individual PEs may be executing different instructions on different pieces of data at any time. This opportunity – also called *thread-level parallelism* – is a heavily used optimization technique in columnar database engines [16, 27, 30]. In detail, MIMD is used to realize a *data-partitioned intra-operator* parallelism. Here, all data objects, e.g., columns, are logically partitioned and partitions are exclusively accessed by the assigned worker thread that is pinned to a specific PE. With this so-called data-oriented architecture [16, 22], the same operator or query pipeline is *logically simultaneously*

```
1  uint64_t const * data; /* previously initialized */
2  uint64_t sum = 0;
3  #pragma omp parallel for num_threads(T_CNT) reduction(+:sum)
4  for (size_t i = 0; i < total_elements; ++i) {
5      const auto val = data[i];
6      sum += val;
7  }
```

Listing 1: Basic OpenMP aggregation.

```
1  #pragma omp declare reduction(simd_add :...)
2  uint64_t const * data; /* previously initialized */
3  uint64_t sum = 0;
4  __m512i vec_sum = _mm512_set1_epi64(0);
5  #pragma omp parallel for num_threads(T_CNT) reduction(simd_add : vec_sum)
6  for (size_t i = 0; i < total_elements / VEC_SZ; ++i) {
7      const auto vals = _mm512_loadu_epi64(data + (i * VEC_CNT));
8      vec_sum = _mm512_add_epi64(vec_sum, vals);
9  }
```

Listing 2: Basic OpenMP aggregation with explicit SIMD.

executed on disjoint data partitions. Data partitions are usually equally sized, so every PE processes the same amount of data, thus minimizing overall runtime.

Listing 1 illustrates our running example implemented as a partitionable loop. The `#pragma` *omp parallel* instructs the compiler to partition the following `for` loop into *T_CNT* partitions (if specified) and consequently assigns a set of loop iterations to a distinct thread. Iterations can be assigned en bloc or interleaved based on a static or dynamic OpenMP scheduler. However, a specific iteration is only processed by a single thread. To avoid data loss or even a segmentation fault, the global `sum` variable must not be written by multiple threads simultaneously Therefore, it could be either made atomic or a thread-local partial sum is calculated, e.g., through the custom `reduction(+:sum)` operation, which only adds up the partial sums at the end of each block.

Therefore, this approach naturally extends the scalar processing to a thread-level parallelism and is widely used as a standard. Especially for query operators with a sequential memory access pattern, no sophisticated control mechanisms are required, as there are usually no data or control flow dependencies between the processed partitions [16, 27, 30]. But it can also be used to realize more complex operators [27]. We can conclude that MIMD is used to realize a *logically synchronous* – but physically asynchronous – and *independent* processing of *logically partitioned data* for an efficient analytical query processing.

2.2 SIMD

Contrary to MIMD, SIMD describes computing units with multiple PEs – SIMD register lanes – that perform the same instruction on multiple data elements parallel in lock-step. That means SIMD exploits *data-level parallelism*, but not concurrency: there are parallel computations, but each PE performs the exact same instruction on different pieces of data at any time. On general-purpose CPUs, each core offers SIMD capabilities through specific SIMD instruction set extensions with varying capabilities (e.g., Intel SSE, AVX2, AVX512, or ARM Neon/SVE). From an abstract point of view, the state-of-the-art SIMD processing for an efficient analytical query processing resembles its scalar counterpart [8, 26, 27]. Data has to be loaded sequentially into a (SIMD-)register, which can then be further processed through explicit intrinsics rather than higher-level abstract operators such as + or -. Lines 7 and 8 of Listing 2 show the exact translation of the scalar variant via AVX512 intrinsics. Based on the SIMD properties and their standard interpretation, we can conclude that SIMD is used to realize a *synchronous* and *dependent* processing of *consecutive* data elements for efficient analytical processing. This is an entirely different approach compared to the thread-level parallelism discussed above.

2.3 State-of-the-art MIMD-SIMD interplay

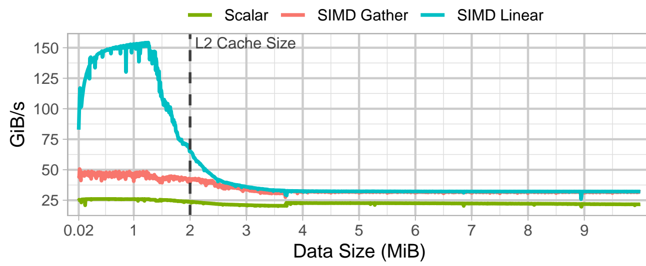
Nevertheless, Listing 2 also shows that linear SIMD processing can be easily enriched with OpenMP to combine both approaches, i.e., *thread-level parallelism* and *data-level parallelism* happen in the same operator. On the outer scope, data is divided into a set of logical partitions, which are processed by an individual thread. However, the SIMD approach processes multiple elements simultaneously in the inner scope per partition. This amalgamation employs two orthogonal algorithmic approaches to realize an intra-operator parallelism, although both want to execute the same operator code for different data elements.

The difference can be explained in more detail from the data access perspective, in this case, from the underlying array of our running example. While non-contiguous array elements are accessed across all *thread-level* PEs, only contiguous array elements are accessed by *data-level* PEs. Moreover, the non-contiguous access is logically equivalent to a strided access, where array elements are accessed equidistantly because the partitions are equally sized. Therefore, the stride distance (or stride size) equals the partition size. However, this strided access is implicitly executed by the thread-level parallelism, as each thread is assigned its own start position in the array and runs through the array from this start position to the end position of the respective data partition.

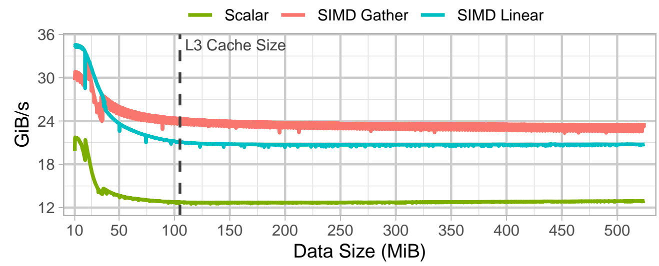
This interleaving of strided and linear access naturally makes implementing parallel query operators more difficult, as different approaches have to be taken into account. To overcome that, we argue that the strided access can also be applied to *data-level parallelism (SIMD)*, which thus allows for a unified parallelization concept as follows: On the outer *thread-level* scope, data is divided into a set of coarse-grained logical partitions, which are processed by an individual thread. On the inner *data-level* scope, coarse-grained logical partitions are further subdivided into fine-grained partitions, which are processed by individual SIMD register lanes. To supplement our claim, we developed a set of carefully designed microbenchmarks, which are explained in the remainder of the paper. Our microbenchmarks tackle different corner cases for combinations of data access patterns, applied query operators, and the underlying storage format.

3 DATA-PARTITIONED SIMD PROCESSING

SIMD extensions of modern general-purpose CPUs consist of two main building blocks: (i) SIMD registers, which are larger than traditional scalar registers, and (ii) SIMD instructions working on those SIMD registers. Contrary to scalar processing, SIMD registers must be explicitly populated with data elements from main memory using a `load` or a `gather` instruction. On the one hand, `load` is

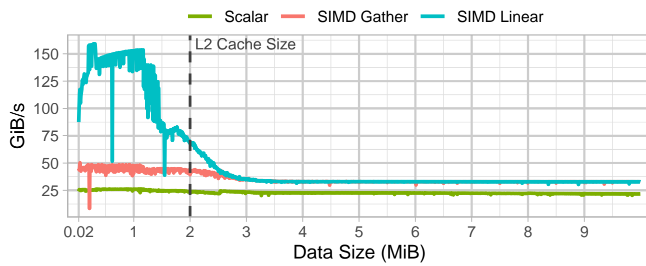


(a) Array Size between 0.02 and 10 MiB.

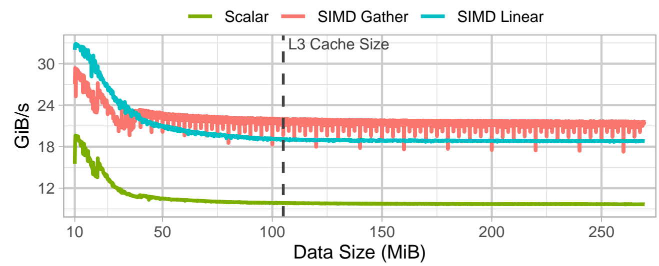


(b) Array Size between 10 and 500 MiB.

Figure 2: Comparing single-threaded throughput for aggregating sum with all data in DDR5 main memory and with different access patterns: scalar, linear, and data-partitioned SIMD.



(a) Array Size between 0.02 and 10 MiB.



(b) Array Size between 10 and 500 MiB.

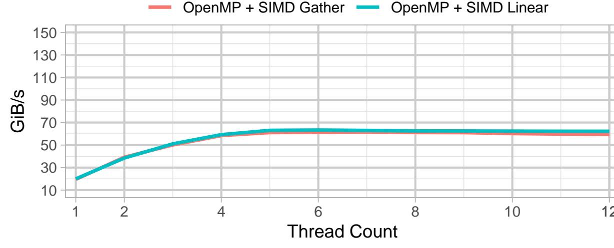
Figure 3: Comparing single-threaded throughput for aggregating sum with all data in HBM main memory and with different access patterns: scalar, linear, and data-partitioned SIMD.

applied, whenever a linear data access pattern is conducted as done in the state-of-the-art for analytical query processing using SIMD capabilities [8, 26, 27]. Linear implies that the accessed data elements are organized as a contiguous sequence like an array. On the other hand, `gather` is used when a random access pattern – data elements from non-consecutive memory locations – is required. The general guideline has been that `gather` should be avoided as far as possible due to the considerable performance loss.

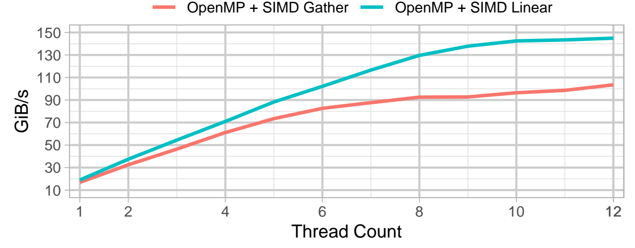
In [13], we already have shown that the `gather` instruction can achieve the same performance as the `load` instruction with a so-called *block-strided access pattern*. A memory access pattern is called strided when memory fields accessed are equally distant and the distance is usually denoted as stride size. The particular property of our proposed block-strided access pattern is that the input data, e.g., an array, is logically divided into blocks. In the simplest case, each block consists of k consecutive pages, where k is the number of SIMD lanes of the underlying SIMD register. The blocks are successively processed and for each block, the SIMD processing works as follows: Each SIMD lane is assigned a page from the block and each lane is further responsible for processing the assigned page. To achieve that, a strided access with the page size as the stride size is performed on the block using the `gather` instruction.

However, the SIMD processing with this block-strided access pattern is quite complicated and requires a two-stage partitioning into blocks as well as pages. To overcome that shortcoming, we investigated the conducted simple partitioning approach of *thread-level parallelism* for SIMD processing. In this case, data is logically divided into k equally-sized partitions in a straightforward way and partitions are exclusively pinned to a specific SIMD lane. To load the corresponding data elements of the k data partitions into the k lanes of a SIMD register, we issue a `gather` instruction conducting a strided access, whereby the stride size now equals the partition size. Then, the same processing functionality through a SIMD instruction is simultaneously executed on the loaded data elements. Subsequently, the consecutive data elements within the k partitions are loaded until all data elements of the partitions are processed.

In the case of our aggregating sum, each SIMD lane computes a partial sum per data partition like each thread but synchronously. The advantage now is that we apply the same instruction independently for each lane and can, therefore, rely on element-wise SIMD instructions. In the final step, we have to add up the individual partial sums, which we can do through the use of horizontal addition. Figure 2 compares the single-threaded throughput results for the aggregating sum operation with all data in DDR5 main memory on our hardware system using (i) purely scalar processing (Scalar), (ii) AVX-512 SIMD processing with a linear access pattern (SIMD Linear), and (iii) AVX-512 SIMD processing with the described simple data-partitioned approach (SIMD Gather). In the experiments, we varied the array size with randomly generated `uint64_t` elements in the range from 0.02 to 500 MiB as depicted on the x-axes of both



(a) Throughput on DRAM.



(b) Throughput on HBM.

Figure 4: Throughput for OpenMP with a linear and data-partitioned aggregating sum on SIMD with different memory types.

diagrams in Figure 2. As we can see, the linear SIMD variant offers significant throughput advantages if the data fits into the L2 cache (2 MiB on our CPU). However, if the total amount of data exceeds the size of the L3 cache, then our proposed data-partitioned SIMD approach outperforms the linear variant. This effect also applies for AVX2 with 32-bit and 64-bit data types but not for AXV512 with 32-bit data types, which is in line with the results presented in [13]. In addition, we repeated the same experiments with all data in HBM2 main memory on our hardware system and Figure 3 shows the measured throughput values. The results confirm the DRAM findings for HBM as well.

4 COLUMNAR STORAGE MODEL

Traditionally, SIMD is employed in analytical database engines whenever a columnar storage or decomposition storage model (DSM) is implemented [8, 26, 27]. Two reasons are decisive for this: (i) only the columns that are relevant to the query need to be read, and (ii) the values per column are stored contiguously and can therefore be processed very well with a linear access pattern. In the previous section, we showed that our *data-partitioned* SIMD processing is on-par or even better than a linear SIMD processing with the limitation to a single-threaded and single-column environment. This section transfers this finding in the multithreaded and the multiple-column environment.

Multi-threaded Environment: Listing 3 and Figure 5 show our proposed unified parallelization concept for thread-level and data-level parallelism illustrated on our aggregating sum example. In the illustration of Figure 5, each of the four threads is assigned

```
1  #pragma omp parallel num_threads(T_CNT) reduction(+:sum)
2  {
3    uint64_t sum = 0;
4    const size_t offset = elements_per_thread / VEC_SZ;
5    const __m512i idx = _mm512_setr_epi64(
6      0, 1 * offset, 2 * offset, ..., 7 * offset);
7    for (size_t i = 0; i < offset; ++i) {
8      const auto vals = _mm512_i64gather_epi64(
9        idx,
10       data + (omp_get_thread_num() * elements_per_thread) + i,
11       sizeof(uint64_t));
12     sum += _mm512_reduce_add_epi64(vals);
13   }
14 }
```

Listing 3: Hierarchical partitoned aggregating sum using OpenMP and strided access.

to a contiguous, coarse-grained and equally-sized partition of the data column. Within each logical partition, we subdivide again into four smaller partitions, whereby each equally-sized fine-grained partition is now processed by one of the register lanes in parallel. To apply the same partitioning scheme, the MIMD stride size equals one-fourth of the data size, whereas the SIMD stride size equals one-fourth of the coarse-grained partition size. Hence, the access pattern for both the global and local computation follows the same pattern. The corresponding code snippet is depicted in Listing 3.

Now, Figure 4 compares the achieved performance results for the aggregating sum of the state-of-the-art OpenMP approach from Listing 2 to our proposed unified data-partitioned approach as outlined in Listing 3. We varied the explicit T_CNT parameter from 1 to 12, since our hardware exhausts the memory bus with 12 concurrent threads in a memory-bound scenario. Further, we tested different data placements for both DRAM and HBM. When the data was placed in DRAM, cf. Figure 4a, we can observe a plateau forming at around 63 GiB/s, for both SIMD-linear and SIMD-Gather. The right-hand side of this figure shows the results of the same experiment but with data placed in HBM. However, no such plateau can be observed for either variant, leaving a throughput gap between the two of about 40 GiB/s. We used the code from the mentioned listings for this experiment and fixed the coarse-grained partition size per thread to 256 MiB, filled with 64-bit unsigned integer values. Consequently, when increasing T_CNT, the total amount of data increases, but the coarse- and fine-grained partition size and, hence, both the outer and inner stride size stays constant. We used AVX512 for SIMD processing, which provides 8 SIMD lanes for 64-bit values. The general finding of this experiment emphasizes our claim that, at least for DRAM, our proposed data-partitioned SIMD processing can be applied to the inner loop part while maintaining a comparable throughput.

Multiple Columns: To analyze the effect of the slightly lower throughput compared to the microbenchmark results from Section 3, we take a step back to the single-threaded execution but

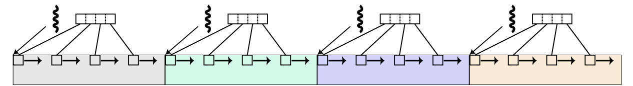
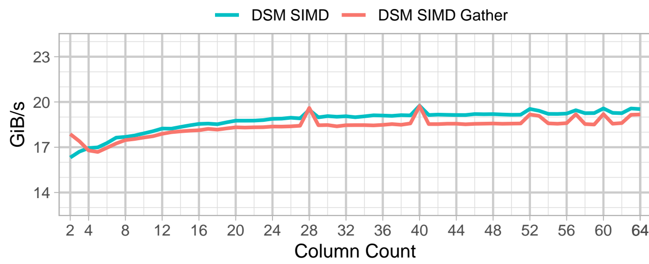
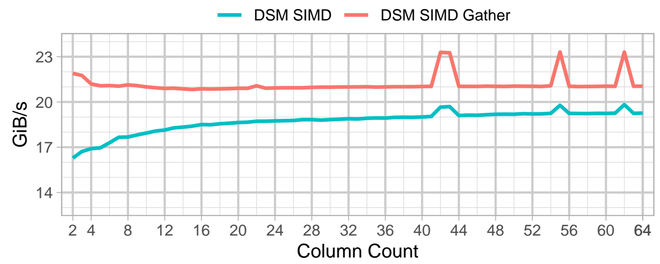


Figure 5: Unified data access pattern for a combined outer MIMD and inner SIMD aggregating sum.

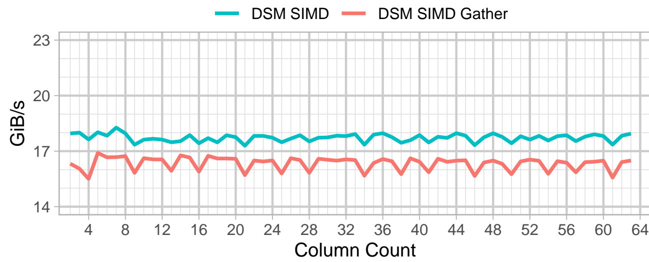


(a) Column Size fixed to 128 MiB.

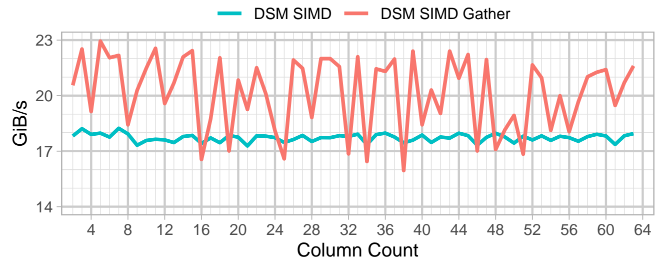


(b) Column Size fixed to 127.98 MiB.

Figure 6: Filter-aggregating sum comparison for linear load and gather, based on a DSM storage layout in DRAM.



(a) Column Size fixed to 128 MiB.



(b) Column Size fixed to 127.98 MiB.

Figure 7: Filter-aggregating sum comparison for linear load and gather, based on a DSM storage layout in HBM.

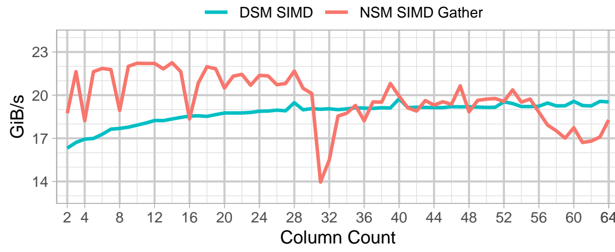
with multiple columns now. We expanded our aggregating sum into a filter-aggregating sum scenario to do this. This scenario works as follows: Assuming we have X columns, then a filter condition is checked on each of the $X - 1$ columns. Only the last attribute is used for the aggregation if the tuple qualifies, and we observe that the selectivity of the filter has no visible impact since we have to read all the data in any case. In Figure 6a, we have chosen a column size of 128 MiB and used this size for all columns. Since we consider AVX512 and 64-bit integer data, our fine-grained partitions have a size of 16 MiB in this case. This size is also our stride size for the necessary strided access. Moreover, each fine-grained partition occupies 4096 pages in this case since our system uses a page size of 4 KiB. This results in a setting where the fine-grained partitions are page-aligned, and therefore, the strided access is also page-aligned. The strided access now loads the following data: The first `gather` instruction loads the elements at position 0 of 8 different pages, whereby the page distance between two elements is 4096 pages (stride size of 16 MiB). The subsequent `gather` instructions load the following elements with ascending positions. As illustrated in Figure 6a, the achieved throughput of the data-partitioned SIMD processing (DSM SIMD Gather) is slightly slower compared to linear SIMD processing (DSM SIMD). This is also consistent with the results as presented in [13].

Based on this finding, however, the question arises as to why the microbenchmark results from Section 3 have shown a performance benefit. The reason for this can only be the page alignment. To investigate that, we reduced the total amount of data by a small fraction in a second experiment, i.e., to 127.98 MiB per column, which in turn means that not every fine-grained partition starts on a new page. As visible in Figure 6b, the linear SIMD processing achieves the same throughput results as for the column size of 128 MiB (cf. Figure 6a). However, now our data-partitioned SIMD processing clearly outperforms the linear variant. The main takeaway from this experiment is, that we achieve higher throughput values compared to the state-of-the-art when the resulting fine-grained partitions are not perfectly page-aligned. We suspect the hash function of the cache system to play a crucial role in this effect since it might hash accessed data elements to conflicting cache positions.

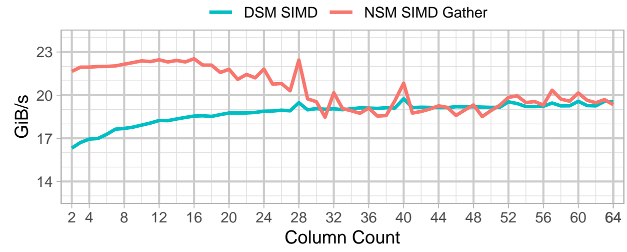
Figure 7 recreates the experiment from Figure 6, but places the data in HBM instead of DRAM. We can observe that the same effects are visible for both memory technologies, but there is more variance when working with HBM. We can conclude that the strided gather access can be employed to achieve comparable or higher performance, but (i) the actual gain is dependent on where the data is actually stored and (ii) the data partition size is important, i.e., if data access is page aligned or not.

5 N-ARY STORAGE MODEL

While the application of SIMD in a DSM environment is considered state-of-the-art, quite the opposite is true for the N-ary Storage Model (NSM), also known as row-store [33]. Generally, using SIMD in an NSM environment is currently ill-advised since the linearly loaded elements contain values from different columns, which (i) do not necessarily contribute to the current operator and (ii) may thus incur an unnecessary burden on the memory system since their values should just be disregarded and discarded.

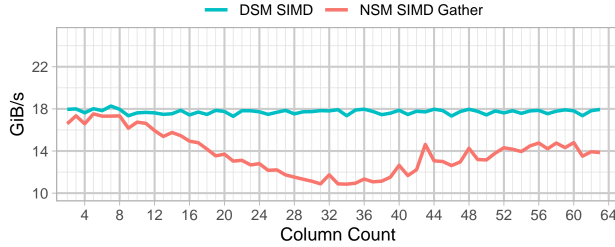


(a) Static Gather access pattern.

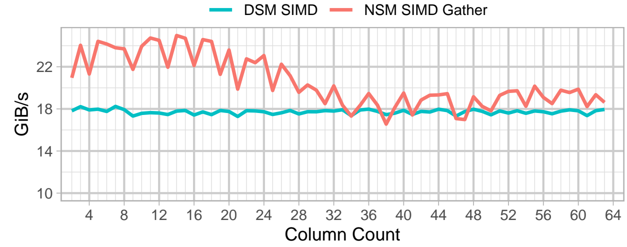


(b) Gather stride size set to 208 elements.

Figure 8: Filter-aggregating sum comparison for linear load and gather, based on an NSM storage layout in DRAM.



(a) Static Gather access pattern.



(b) Gather stride size set to 208 elements.

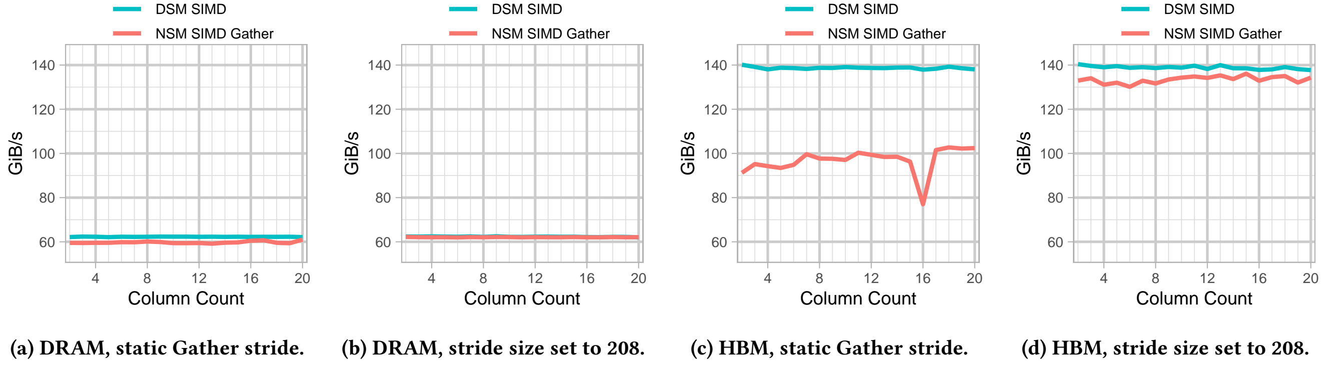
Figure 9: Filter-aggregating sum comparison for linear load and gather, based on an NSM storage layout in HBM.

Since we can efficiently load data from different locations with our proposed data-partitioned strided access, this should actually reduce the challenge to some degree. To investigate that aspect in more detail, we executed the same benchmark from Figure 6, but with an NSM storage model. For NSM, we created a large array for each column count and saved the tuples consecutively with the attribute values for each tuple one after the other (array-of-struct). According to our data-partitioned SIMD processing concept, we divide the large array into eight equally-size partitions for AVX512 and 64-bit integer values. Then, processing can take place column-wise by loading 8 column values from 8 different tuples. Figure 8a shows the achieved throughput results for the data-partitioned NSM SIMD processing in comparison to the state-of-the-art DSM SIMD processing (linear variant) if we partition the NSM data as-is. This, of course, leads to the page-alignment problem as discussed above. However, this can be addressed by choosing a fine-grained partition size that does not lead to this problem, as shown in Figure 8b. For this diagram, we tried out different fine-grained partition sizes and selected the best one. The remaining tuples that were not assigned to a partition were subsequently processed as scalar remainder. We repeated the same experiment for HBM and presented the results in Figure 9a. It shows, that the same page-alignment problem occurs, which in this case leads to a completely worse performance of our NSM concept compared to the state-of-the-art. When we select the same partition size as previously, we can improve the throughput again, as seen in Figure 9a. The main takeaway from these experiments is, that we can apply our data-partitioned SIMD processing for NSM as well and achieve similar throughput results as for DSM. Moreover, we can optimize our data-partitioned SIMD processing by carefully selecting an appropriate fine-grained partition size, which works both for DRAM and HBM.

We further tested the applicability of our approach in an MIMD environment. Hence, we assign each thread the same amount of data to mimic the algorithmic behavior as closely as possible and keep the gathered strides equal to the scalar case. For DRAM, the threads would each process 128 MiB per column. However, using the same column size in HBM would only permit 5 columns, which is an amount that is too small to point to a trending behavior. Therefore, we reduced the data per column to 32 MiB, which allowed us to process up to 20 columns with 12 concurrent threads.

Figure 10a shows that NSM SIMD Gather is around 3 GiB/s slower compared to DSM SIMD, 59.5 GiB/s to 62 GiB/s. When adjusting the partition size, this throughput gap gets closed (cf. Figure 10b). For HBM this gap is much more noticeable, roughly 50 GiB/s, and again, nearly disappears when we optimize the partition size, ending up in a difference of around 6 GiB/s (cf. Figure 10c and Figure 10d). The main takeaway from this experiment is that the previously fine-grained partition size, not only optimizes single threaded performance but also multithreaded performance. Overall, our data-partitioned SIMD processing is competitive in DRAM as is and with a carefully selected partition size even in HBM.

A limiting factor of the NSM SIMD Gather approach is that columns are unused during query processing. With an NSM approach, every column is loaded whether it contributes to the query or not, thus wasting precious bandwidth. This effect is illustrated in Figure 11, which shows that the throughput drops for NSM SIMD Gather with a decreasing percentage of attributes used. Expectedly,



(a) DRAM, static Gather stride. (b) DRAM, stride size set to 208. (c) HBM, static Gather stride. (d) HBM, stride size set to 208.

Figure 10: Filter-aggregating sum comparison for linear load and gather, 12 Threads.

both DSM experiments can maintain their respective bandwidth since only the necessary data is read.

However, processing larger vectors, either in NSM or DSM layout, is a contemporary use case, e.g., in data clustering, Large Language Models (LLMs) or other AI related research. To validate our claim, that using our unified parallelization concept (MIMD + SIMD) can be beneficial, we chose the calculation of the Manhattan distance, which is commonly used as a distance metric for data with high dimensionality. Figure 12 compares the calculation of the Manhattan distance using linear SIMD based on the DSM layout from Section 4 with a data-partitioned SIMD implementation which is based on an NSM layout as shown in Section 5. The experiment is set in a multithreaded environment with 12 parallel threads to investigate the MIMD-SIMD interplay. We deliberately chose the seemingly best-case state-of-the-art, i.e., linear SIMD on columns, to compare against the unconventional gather on a row store with an ideal partition size. For this experiment, we again set the data size per column and per thread to 128 MiB for DRAM (Figure 12a) and 32 MiB for HBM respectively (Figure 12b). Our observation can be explained as follows. Consider a reference vector $\overrightarrow{v_r} = \{x, y, z\}$ and the three respective columns $\overrightarrow{X}, \overrightarrow{Y}, \overrightarrow{Z}$ containing the dimensional values for multiple vectors $\overrightarrow{V}$. In the DSM format, we calculate the Manhattan distance of $\overrightarrow{v_r}$ to all vectors $\overrightarrow{V}$ using the following scheme:

(0) Initialize a temporary buffer b that fits into L1d with 0, broadcast x from $\overrightarrow{v_r}$ into a SIMD register $\overrightarrow{r_{dim}}$.
(1) Load a SIMD register from a dimension-column, e.g., $\overrightarrow{v_x} = X[i : i{+}VEC_SZ]$ and load the corresponding elements from b into another register $\overrightarrow{v_b}$.
(2) Calculate the absolute difference between $\overrightarrow{r_{dim}}$ and $\overrightarrow{v_x}$ and add it to $\overrightarrow{v_b}$.
(3) Store $\overrightarrow{v_b}$ back into the buffer b, overwriting the previous values.
(4) If the buffer is full, broadcast the next dimension from $\overrightarrow{v_x}$ to $\overrightarrow{r_{dim}}$ and start over from step (1) with the next column.
(5) When all columns are calculated, determine the minimum value within b and start over from step (0) for the remaining elements in the columns.

In contrast, when using the NSM layout and strided gather access, the algorithm works as follows:

(0) Initialize a SIMD register $\overrightarrow{v_{res}}$ with 0, set i to 0 and broadcast x into a SIMD register $\overrightarrow{r_{dim}}$.
(1) Gather unprocessed elements from a column, e.g., $\overrightarrow{v_x} = \{X[i], X[i+N], X[i+2N], ...\}$.
(2) Calculate the absolute difference between $\overrightarrow{v_x}$ and $\overrightarrow{v_{dim}}$ and add it to $\overrightarrow{v_{res}}$.
(3) Increment i by 1 and start over from (1).

Besides the different access strategies and data layouts, both algorithms vary primarily in how intermediates are treated. While the DSM variant relies on a temporary buffer, the NSM variant directly processes the data in registers. As shown in Figure 12a, the DSM is memory-bound and the temporary buffer does not introduce relevant overhead. We argue that with a relative difference of approximately 1%, the performance of DSM and NSM is on par. Interestingly, NSM, in combination with our novel approach, obtains up to 10% higher throughput on HBM compared to the DSM variant. The observable offset is likely to stem from the fact that the DSM variant incurs two load operations from HBM per iteration, which suffers from the additional latency of HBM and thus underutilizes the available memory subsystem.

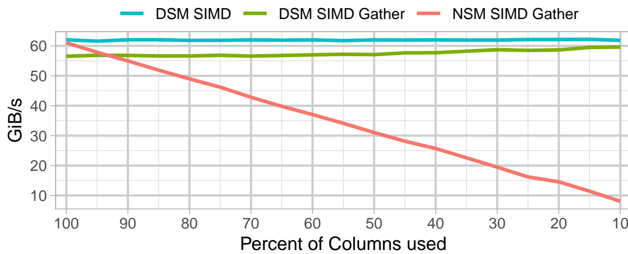
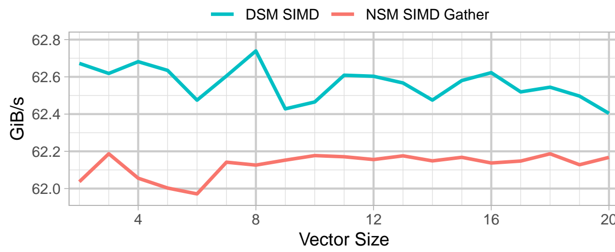


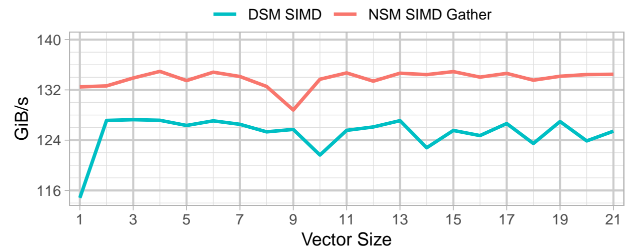
Figure 11: Throughput for varying percent of attributes used.

6 CONCLUSION AND FUTURE WORK

Exploiting parallelism is crucial to achieve low latency for analytical queries in in-memory database engines. For example, modern general-purpose CPUs offer high-computational power through two data-oriented parallel paradigms: MIMD and SIMD. As both



(a) Columns in DRAM with 128 MiB per Column per Thread.



(b) Columns in HBM with 32 MiB per Column per Thread.

Figure 12: Throughput for calculating the minimum Manhattan distance using 12 Threads. Beware of different data amounts and scaling of y-axes.

tackle different granularities, data has to be partitioned differently to satisfy their respective requirements, which in turn creates algorithmic overhead when combining them. To overcome that, we showed that we are able to seamlessly transfer the MIMD partitioning to SIMD in this paper due to the recent advances for SIMD on CPUs. Moreover, we clearly demonstrated that the resulting unified parallelism approach offers a number of advantages for DSM and NSM data layouts, e.g., the performance is similar to the state-of-the-art and even outperforms it in many cases.

From our perspective, our proposed approach offers interesting points for future work. On the one hand, the transfer of MIMD techniques for SIMD should be further investigated. Our approach can only be seen as an initial attempt showing the potential. On the other hand, it would of course also be very interesting to investigate whether the simple OpenMP programming approach for MIMD can also be transferred to SIMD. This would (i) simplify the SIMD utilization and (ii) improve the possibilities for autovectorization.

ACKNOWLEDGMENT

This work was partly funded by (1) the German Research Foundation (DFG) priority program SPP 2377 under grant no. LE 1416/30-1, (2) the European Union's Horizon 2020 research and innovation program under grant agreement no. 957407 (DAPHNE), and (3) the German Research Foundation (DFG) via a Reinhart Koselleck-Project (LE-1416/28-1).

REFERENCES

[1] Anastasia Ailamaki, Erietta Liarou, Pinar Tözün, Danica Porobic, and Iraklis Psaroudakis. 2014. How to stop under-utilization and love multicores. In *International Conference on Management of Data, SIGMOD 2014, Snowbird, UT, USA, June 22-27, 2014*. ACM, 189–192. https://doi.org/10.1145/2588555.2588892

[2] Joy Arulraj, Andrew Pavlo, and Subramanya Dulloor. 2015. Let's Talk About Storage & Recovery Methods for Non-Volatile Memory Database Systems. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015*. ACM, 707–722. https://doi.org/10.1145/2723372.2749441

[3] André Berthold, Lennart Schmidt, Antonia Obersteiner, Dirk Habich, Wolfgang Lehner, and Horst Schirmeier. 2024. On-The-Fly Data Distribution to Accelerate Query Processing in Heterogeneous Memory Systems. In *Advances in Databases and Information Systems - 28th European Conference, ADBIS 2024, Bayonne, France, August 28-31, 2024, Proceedings (Lecture Notes in Computer Science, Vol. 14918)*. 170–183. https://doi.org/10.1007/978-3-031-70626-4_12

[4] Peter A. Boncz, Martin L. Kersten, and Stefan Manegold. 2008. Breaking the memory wall in MonetDB. *Commun. ACM* 51, 12 (2008), 77–85.

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

[6] Shekhar Borkar and Andrew A. Chien. 2011. The future of microprocessors. *Commun. ACM* 54, 5 (2011), 67–77. https://doi.org/10.1145/1941487.1941507

[7] Sebastian Breß, Henning Funke, and Jens Teubner. 2016. Robust Query Processing in Co-Processor-accelerated Databases. In *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*. 1891–1906. https://doi.org/10.1145/2882903.2882936

[8] Patrick Damme, Annett Ungethüm, Johannes Pietrzyk, Alexander Krause, Dirk Habich, and Wolfgang Lehner. 2020. MorphStore: Analytical Query Engine with a Holistic Compression-Enabled Processing Model. *Proc. VLDB Endow.* 13, 11 (2020), 2396–2410. http://www.vldb.org/pvldb/vol13/p2396-damme.pdf

[9] Jaeyoung Do, Yang-Suk Kee, Jignesh M. Patel, Chanik Park, Kwanghyun Park, and David J. DeWitt. 2013. Query processing on smart SSDs: opportunities and challenges. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2013, New York, NY, USA, June 22-27, 2013*. 1221–1230. https://doi.org/10.1145/2463676.2465295

[10] Franz Färber, Sang Kyun Cha, Jürgen Primsch, Christof Bornhövd, Stefan Sigg, and Wolfgang Lehner. 2011. SAP HANA database: data management for modern business applications. *SIGMOD Rec.* 40, 4 (2011), 45–51. https://doi.org/10.1145/2094114.2094126

[11] Michael J. Flynn. 1972. Some Computer Organizations and Their Effectiveness. *IEEE Trans. Computers* 21, 9 (1972), 948–960. https://doi.org/10.1109/TC.1972.5009071

[12] Dirk Habich and Johannes Pietrzyk. 2024. SIMDified Data Processing - Foundations, Abstraction, and Advanced Techniques. In *Companion of the 2024 International Conference on Management of Data, SIGMOD/PODS 2024, Santiago AA, Chile, June 9-15, 2024*. ACM, 613–621. https://doi.org/10.1145/3626246.3654694

[13] Dirk Habich, Johannes Pietrzyk, Alexander Krause, Juliana Hildebrandt, and Wolfgang Lehner. 2022. To use or not to use the SIMD gather instruction?. In *International Conference on Management of Data, DaMoN 2022, Philadelphia, PA, USA, 13 June 2022*, Spyros Blanas and Norman May (Eds.). ACM, 9:1–9:5. https://doi.org/10.1145/3533737.3535089

[14] Wenqi Jiang, Dario Korolija, and Gustavo Alonso. 2023. Data Processing with FPGAs on Modern Architectures. In *Companion of the 2023 International Conference on Management of Data, SIGMOD/PODS 2023, Seattle, WA, USA, June 18-23, 2023*. ACM, 77–82. https://doi.org/10.1145/3555041.3589410

[15] Tomas Karnagel, Dirk Habich, and Wolfgang Lehner. 2017. Adaptive Work Placement for Query Processing on Heterogeneous Computing Resources. *PVLDB* 10, 7 (2017), 733–744.

[16] Thomas Kissinger, Dirk Habich, and Wolfgang Lehner. 2018. Adaptive Energy-Control for In-Memory Database Systems. In *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*. ACM, 351–364. https://doi.org/10.1145/3183713.3183756

[17] Alberto Lerner, Carsten Binnig, Philippe Cudré-Mauroux, Rana Hussein, Matthias Jasny, Theo Jepsen, Dan R. K. Ports, Lasse Thostrup, and Tobias Ziegler. 2023. Databases on Modern Networks: A Decade of Research that now comes into Practice. *Proc. VLDB Endow.* 16, 12 (2023), 3894–3897. https://doi.org/10.14778/3611540.3611579

[18] Feng Li, Sudipto Das, Manoj Syamala, and Vivek R. Narasayya. 2016. Accelerating Relational Databases by Leveraging Remote Memory and RDMA. In *SIGMOD*. 355–370.

[19] 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. https://doi.org/10.14778/3151113.3151114

[20] Thomas Neumann. 2011. Efficiently Compiling Efficient Query Plans for Modern Hardware. *Proc. VLDB Endow.* 4, 9 (2011), 539–550. https://doi.org/10.14778/2002938.2002940

[21] Ismail Oukid and Wolfgang Lehner. 2017. Data Structure Engineering For Byte-Addressable Non-Volatile Memory. In *Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD Conference 2017, Chicago, IL, USA, May 14-19, 2017*. ACM, 1759–1764. https://doi.org/10.1145/3035918.3054777

[22] Ippokratis Pandis, Ryan Johnson, Nikos Hardavellas, and Anastasia Ailamaki. 2010. Data-Oriented Transaction Execution. *Proc. VLDB Endow.* 3, 1 (2010), 928–939. https://doi.org/10.14778/1920841.1920959

[23] Johannes Pietrzyk, Alexander Krause, Christian Faerber, Dirk Habich, and Wolfgang Lehner. 2024. Program your (custom) SIMD instruction set on FPGA in C++. In *14th Conference on Innovative Data Systems Research, CIDR 2024, Chaminade, HI, USA, January 14-17, 2024*. www.cidrdb.org. https://www.cidrdb.org/cidr2024/papers/p53-pietrzyk.pdf

[24] Constantin Pohl, Kai-Uwe Sattler, and Goetz Graefe. 2020. Joins on high-bandwidth memory: a new level in the memory hierarchy. *VLDB J.* 29, 2-3 (2020), 797–817. https://doi.org/10.1007/S00778-019-00546-Z

[25] Fred J. Pollack. 1999. New Microarchitecture Challenges in the Coming Generations of CMOS Process Technologies. In *Proceedings of the 32nd Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 32, Haifa, Israel, November 16-18, 1999*. ACM/IEEE Computer Society, 2. https://doi.org/10.1109/MICRO.1999.10004

[26] 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*. ACM, 1493–1508. https://doi.org/10.1145/2723372.2747645

[27] Orestis Polychroniou and Kenneth A. Ross. 2020. VIP: A SIMD vectorized analytical query engine. *VLDB J.* 29, 6 (2020), 1243–1261. https://doi.org/10.1007/S00778-020-00621-W

[28] Wolf Rödiger, Tobias Mühlbauer, Alfons Kemper, and Thomas Neumann. 2015. High-Speed Query Processing over High-Speed Networks. *Proc. VLDB Endow.* 9, 4 (2015), 228–239. https://doi.org/10.14778/2856318.2856319

[29] Yuan Xie and Jishen Zhao. 2019. Emerging Memory Technologies. *IEEE Micro* 39, 1 (2019), 6–7. https://doi.org/10.1109/MM.2019.2892165

[30] Mikhail Zarubin, Patrick Damme, Alexander Krause, Dirk Habich, and Wolfgang Lehner. 2021. SIMD-MIMD cocktail in a hybrid memory glass: shaken, not stirred. In *SYSTOR '21: The 14th ACM International Systems and Storage Conference, Haifa, Israel, June 14-16, 2021*. ACM, 17:1–17:12. https://doi.org/10.1145/3456727.3463782

[31] Hao Zhang, Gang Chen, Beng Chin Ooi, Kian-Lee Tan, and Meihui Zhang. 2015. In-Memory Big Data Management and Processing: A Survey. *IEEE Trans. Knowl. Data Eng.* 27, 7 (2015), 1920–1948. https://doi.org/10.1109/TKDE.2015.2427795

[32] Tobias Ziegler, Carsten Binnig, and Viktor Leis. 2022. ScaleStore: A Fast and Cost-Efficient Storage Engine using DRAM, NVMe, and RDMA. In *SIGMOD '22: International Conference on Management of Data, Philadelphia, PA, USA, June 12 - 17, 2022*. 685–699. https://doi.org/10.1145/3514221.3526187

[33] Marcin Zukowski, Niels Nes, and Peter A. Boncz. 2008. DSM vs. NSM: CPU performance tradeoffs in block-oriented query processing. In *4th Workshop on Data Management on New Hardware, DaMoN 2008, Vancouver, BC, Canada, June 13, 2008*, Qiong Luo and Kenneth A. Ross (Eds.). ACM, 47–54. https://doi.org/10.1145/1457150.1457160