To stride or not to stride the memory access?

Anonymous Author(s)

Abstract

Due to the increasing gap between CPU performance and memory bandwidth, memory access patterns play more and more a significant role for an efficient data processing. The current core assumption is that a sequential access pattern delivers the best performance, especially when the data to be processed is stored in adjacent memory locations (contiguous memory). Given the continuous advances in memory technologies, it is of course questionable whether this assumption still holds true. To answer this question, we present a comprehensive experimental comparison of the sequential and the strided access pattern for data stored in contiguous memory on modern disruptive memory systems in this paper. As we are going to show, the core assumption must be revised, as the strided access pattern with a well-chosen stride size clearly outperforms the sequential access pattern. Even a SIMD-accelerated sequential access is considerably slower than the best-performing scalar strided access. In our in-depth analysis, we explain the differences and highlight further advantages of the strided access pattern.

ACM Reference Format:
Anonymous Author(s). 2025. To stride or not to stride the memory access?. In *Proceedings of Workshop on Disruptive Memory System @ SOSP'25 (DIMES'25).* ACM, New York, NY, USA, 7 pages. https://doi.org/XXXXXXX.XXXXXXX

1 Introduction

The key objective of database systems is to reliably manage data, where high query throughput and low query latency are core requirements [1]. To satisfy these requirements, database systems constantly adapt to novel hardware features [2, 4–6, 8, 11, 13, 15, 20]. Therefore, it is not surprising that current memory developments such as non-uniform memory access (NUMA), high-bandwidth memory (HBM) or remote memory designs such as RDMA or CXL are being intensively researched to exploit their specific properties for an efficient data processing [9, 11, 13, 16]. Moreover, these developments are accompanied by adjustments to the entire memory subsystem to enable an efficient access to data in different memory types. This is particularly noticeable on Intel processors. For example, the 4th generation of the Intel Xeon Scalable processor architecture (Sapphire Rapids) not only offers HBM, but also a significant overhaul of the cache structure in terms of size and associativity compared to previous generations like the older 2nd generation (Cascade Lake). In addition, the 4th generation includes improvements to the structure of the Translation Lookaside Buffer (TLB).

Our Contribution and Outline: These memory subsystem adjustments are very interesting, but the impact on the different memory access patterns are only marginally known [3, 7]. According to the state-of-the-art, it is still assumed that a sequential access pattern provides the best performance, especially if the data to be processed is stored in adjacent memory locations (contiguous memory) [4, 17]. In this paper, we show that this assumption no longer applies for data stored in contiguous memory and that a strided access pattern with a well-chosen stride size clearly outperforms sequential access patterns. A strided access pattern is a variant of the sequential access pattern that realizes an equidistant data access, i.e., there is a constant (but configurable) distance – greater than 1 – between accessed data elements in a contiguous sequence. As data is now retrieved from different non-consecutive positions in memory, it is surprising that this strided access pattern can be more efficient than a sequential access pattern. To present our insight, the remainder of the paper is structured as follows:

- In Section 2, we present the detailed evaluation setup and the overall results of our comparative evaluation of the sequential and strided access patterns.
- Then, we deepen the analysis of the strided access pattern by an in-depth investigation of several influencing factors from the CPU side in Section 3.
- In Section 4, we evaluate and discuss the impact of our obtained insights on different memory types, such as HBM and remote memory.

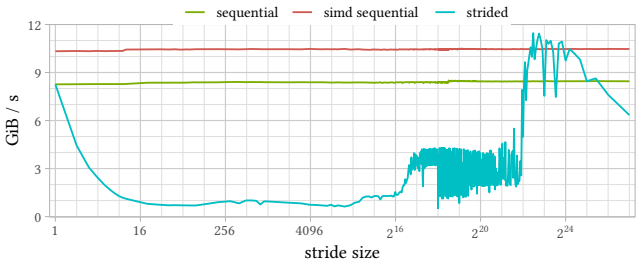
The paper concludes with related work (Section 5) and a summary including ongoing research activities (Section 6).

2 Access Pattern Comparison

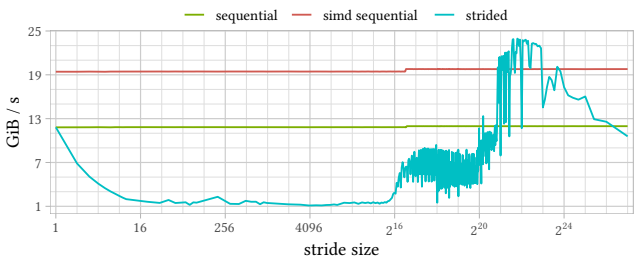
To compare sequential with strided access patterns, we conducted a systematic evaluation of the `aggregation sum` (`AggSum`) operation, because the performance of this operation is mainly dominated by the memory performance and therefore the employed access pattern. `AggSum` computes the aggregating sum over a large column or array of integer data (contiguous memory area). We focused our comparison on

DIMES'25, October 13th, 2025, Seoul, Republic of Korea

ACM ISBN 978-1-4503-XXXX-X/18/06
https://doi.org/XXXXXXX.XXXXXXX



(a) Cascade Lake Architecture.



(b) Sapphire Rapids Architecture.

Figure 1. Single-threaded throughput results for AggSum over an 1 GiB array of uint64_t values.

the following variants: **(1)** ***sequential*****:** AggSum will be computed by sequentially iterating over the array and adding up the elements one after the other. **(2)** ***simd-sequential*****:** This variant is an explicit SIMDified version of the *sequential* variant (using 512-bit AVX512 intrinsics). **(3)** ***strided*****:** AggSum using a strided access pattern as depicted in Listing 1. The stride size is denoted as s, which also determines the number of required passes over the data (outer loop). In each pass, only every s^{th} array element is accessed (inner loop) with a shifted starting position. For example, with a stride size of 2, two passes over the array are necessary (outer loop). In the first pass, the even array positions (0, 2, 4, ...) are sequentially accessed (half of the elements), while in the second pass, the odd positions (1, 3, 5, ...) are sequentially accessed (other half of the data). **(4)** ***strided-unrolled*****:** The last variant is an optimized version of the *strided* variant, where the inner loop is explicitly unrolled. Especially for very large stride sizes, where the inner loop only accesses very few array elements in each pass, unrolling should provide an additional speed advantage over the *strided* variant.

Evaluation Setup. We conducted our evaluation with two generations of Intel Xeon Scalable CPUs, as shown in Table 1. The newer generation (Sapphire Rapids architecture) features 64 GiB of HBM, as well as larger cache sizes and higher cache associativity compared to the older Cascade Lake architecture. Moreover, the Data Translation Lookaside Buffer (DTLB) and the Second-level Translation Lookaside Buffer (STLB) are larger on the newer generation. All AggSum variants were implemented in C++ and compiled using g++ with the optimization flags `-O3 -march=native -mavx512f` as well as `-fno-tree-vectorize` to disable autovectorization, so that the variants *sequential*, *strided*, and *strided-unrolled* were executed in a scalar fashion. We evaluated all variants single-threaded and for different data types, such as `uint64_t` or `uint32_t`. In this paper, we only present

```
1  uint64_t* data; /* assume data array is initialized */
2  uint64_t sum = 0;
3  for (size_t i = 0; i < s; i++) {
4      for (size_t j = 0; j < total_elements/s; j++) {
5          sum += data[i + j * s];
6      }
7  }
```

Listing 1. AggSum using a strided access pattern.

Table 1. System specification comparison.

Architecture	Cascade Lake	Sapphire Rapids
Generation	2^{nd} Xeon Scalable	4^{th} Xeon Scalable
Name	Gold 6240R	Gold 9468
RAM	DDR4 - 192GB -	DDR5 - 256GB HBM - 64GB
L1D Cache	32KiB (8-way)	48KiB (12-way)
L2 Cache	1MiB (16-way)	2MiB (16-way)
L3 Cache	35.75MiB (11-way)	105MiB (15-way)
DTLB	64 Entries	96 Entries
STLB	1536 Entries	2048 Entries

the results for `uint64_t`, as we observed similar effects for `uint32_t`. All experiments happened entirely in-memory with an input array containing 1 GiB of randomly generated `uint64_t` values being allocated with `aligned_alloc`. Based on a 4 KiB page size granularity, the array comprises 262.144 memory pages. All experiments were repeated 11 times with governor on performance and we report the mean results. Since modern Intel CPUs feature sophisticated hardware data prefetching mechanisms, we do not use software-based prefetching [10, 14].

Evaluation Result A representative excerpt of our single-threaded evaluation results for an 1 GiB array of `uint64_t` values is illustrated in Figure 1. In both diagrams, the stride size with a logarithmic scale is shown on the x-axis and the throughput in GiB/s on the y-axis. In our experiments, we explicitly pinned the execution of all AggSum variants on a core of NUMA node 0 and placed the data array in the locally attached DRAM of NUMA node 0 using `numactl`. Since the array contained 2^{27} `uint64_t` elements, we evaluated all possible stride sizes from 1 to 2^{26} in terms of number of elements. From our results, we can derive the following general observations, which are not surprising at all: **(a)** As expected, the *simd-sequential* variant achieves a higher throughput than the *sequential* variant on both generations. This confirms the state-of-the-art understanding that operations based on a sequential access pattern can be optimized using SIMD. **(b)** A strided access with a stride size of 1 (in terms of elements) corresponds to a sequential access pattern resulting in equal throughput values as visible on both generations. As stride

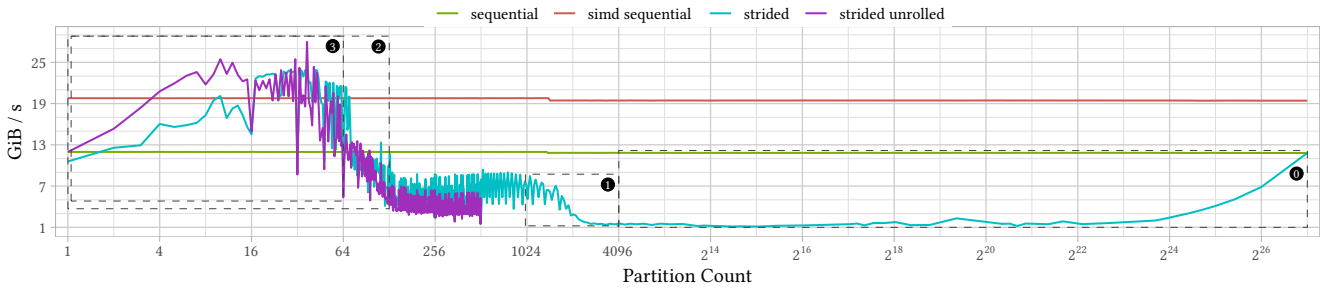


Figure 2. Single-threaded throughput results for `AggSum` over an 1 GiB array of `uint64_t` values on Intel Sapphire Rapids.

size increases, throughput decreases continuously, reflecting current understanding as well.

However, both diagrams in Figure 1 also reveal a *surprising insight*: With increasing large stride sizes (around the stride size of 2^{16}), the throughput of the *strided* variant suddenly increases in several steps. Moreover, the *strided* variant achieves a similar throughput as the *simd-sequential* variant with a stride size of $2^{22.5}$ on the older Cascade Lake architecture on the one hand. On the other hand, the *strided* variant clearly outperforms the *simd-sequential* variant on the newer Sapphire Rapids architecture. While *simd-sequential* only achieves a throughput of 19.8 GiB/s, the strided variant achieves – in the best – a much higher and remarkable throughput of roughly 24 GiB/s for a stride size of $2^{21.8}$.

To explain the throughput behavior of the *strided* variant, we slightly adapt the interpretation: The strided access logically divides the data array into $\frac{n}{s}$ partitions, where (a) n is the total number of array elements and (b) s is the stride size. Moreover, each partition consists of exactly s consecutive elements. In the i^{th} strided pass over the data (outer loop in Listing 1), the i^{th} element per partition is processed. Based on that interpretation, Figure 2 shows our measured throughput results for the different `AggSum` variants on the newer Sapphire Rapids architecture. While a small stride size leads to a high partition count, large stride sizes result in small partition counts. As depicted in Figure 2, now the interesting behavior occurs for small partition counts. Figure 2 also includes the results for the *strided-unrolled* variant with unrolling only up to a partition count of 512. Again, both strided variants clearly outperform the *sequential* as well as the *simd-sequential* variant for small partition counts (or large stride sizes). Moreover, it can be seen that unrolling the inner loop of the strided access (cf. Listing 1) for small partition counts provides a significant throughput boost.

3 In-depth Analysis

To analyze the throughput behavior of the *strided* variants in detail, the starting point is Figure 2 including the setup from Section 2. Generally, the following applies to all access patterns: When an element of the data array is accessed, a cache line is always loaded into the caches, which contains a copy of the memory around this element. A cache line is 64 bytes in size – corresponding to 8 `uint64_t` values – on our Intel (x86) systems. Moreover, if the memory subsystem detects a sequential access pattern, the hardware prefetchers will also prefetch the next cache line (or two) to boost the access to the adjacent data elements.

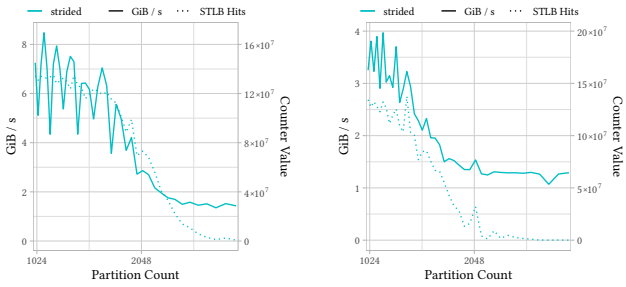
Based on that, the memory subsystem is hardly stressed by our *sequential* variants, because both variants iterate once over the complete array and all data elements are processed in sequence. This means the memory subsystem is only concerned with the sequential prefetching of neighboring cache lines through the different pages. Then, all elements of the cache lines are aggregated by the CPU and here, the *sequential* and the *simd-sequential* variants differ. While the *sequential* variant executes 8 arithmetic instructions per cache line, the *simd-sequential* variant only execute one arithmetic SIMD instruction per cache line leading to a higher throughput as shown in Figure 2.

In contrast, the strided variants heavily stress the memory subsystem as data elements from different non-consecutive memory positions in memory have to be loaded in sequential manner. As shown in Figure 2, this stress can have a very positive and a very negative effect on the throughput and this can be explained using four main observations that we have marked and highlighted in Figure 2:

Observation ⓪

For large numbers of partitions (> 4096), the throughput of the *strided* variant is very low, but smoothly increases for very large numbers of partitions (>2^{24}) until finally reaching the throughput of the *sequential* variant.

The partition count also represents the number of non-consecutive data elements that have to be sequentially loaded per strided pass over the array. For each data element, (i) the virtual address has to be translated into a physical address and (ii) the corresponding cache line has to be fetched. Both tasks overload the memory subsystem for larger partition counts, as each data element requires a disjoint page to be accessed. However, when the number of partitions becomes very large (e.g., > 2^{24} partitions), multiple partitions can then be found within a single page. Furthermore, since our



(a) Sapphire Rapids System. **(b)** Cascade Lake System.

Figure 3. Excerpt of the throughput results for the partition count range from 1024 to 4096.

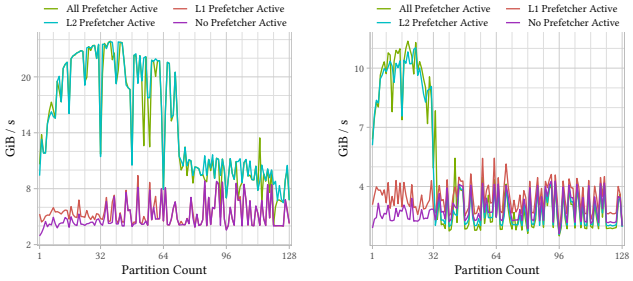
`AggSum` operation operates on 1 GiB of data, with 2^{24} partitions, each partition corresponds to exactly one cache line. This means that if the number of partitions further increases, more than one partition will now reside within one cache line. Since the partitions are accessed sequentially (see the inner loop of Listing 1), accessing multiple partitions now results in multiple accesses to the same cache line. We attribute the increasing throughput with a very large number of partitions (see Figure 2) to these effects.

Observation 1

In the partition count range from 1024 to 4096, a first increase in throughput for the strided access can be observed on both examined Intel CPUs.

Influence of the TLB. Figure 3 shows a zoom-in into the throughput curves for the strided `AggSum` variant within the partition count range from 1024 to 4096 for both examined CPUs. In both diagrams in Figure 3, we can clearly observe a performance increase on both systems once the number of partitions is less than the combined capacity of both TLBs. The increase point is in a different place on both systems, as the examined systems have different TLB configurations for 4 KiB pages (cf. Table 1). Once the performance increases, we can clearly observe an increase in STLB hits as well as a decreased number of triggered page walks that fetch the translation from the system's page table. This observation can be confirmed by using huge pages. With page sizes of 2 MiB and 1 GiB, a performance degradation cannot be observed, since the TLB sizes on both systems are now sufficient to hold all required translations. Therefore, we conclude that an important influencing factor for the strided access pattern is the size of the Translation Lookaside Buffer (TLB), since multiple pages may need to be accessed.

Take away: When employing a strided access pattern, the configuration and the capacity of the TLBs limit the number of partitions to avoid significant performance degradations.



(a) Intel Sapphire Rapids **(b)** Intel Cascade Lake

Figure 4. Excerpt of the throughput results for the partition count range from 1 to 64 with hardware prefetchers enabled and disabled.

Observation 2

A second significant increase in throughput on our systems can be observed for a small number of partitions (< 64), which even exceeds the throughput of the *simd-sequential* variant.

Influence of Hardware Prefetchers We attribute the observed throughput increase primarily to the advanced hardware prefetching mechanisms. Modern CPU architectures are typically equipped with sophisticated prefetching mechanisms embedded into the hardware that detect memory access patterns and proactively load data into the CPU cache before it is explicitly requested. These hardware prefetchers can not only identify simple ascending or descending address patterns, but are also capable of tracking multiple independent memory streams simultaneously. Unfortunately, exact implementation details of the hardware prefetchers are often not publicly available. Both of our systems feature L1 hardware prefetchers that observe memory accesses and can trigger additional prefetchers embedded in the L2 cache. To better understand their impact on throughput, we conducted a systematic evaluation by selectively enabling and disabling the individual prefetchers at both levels of the cache hierarchy. As shown in Figure 4, turning off the L1 prefetchers has only a minor impact on the overall performance. However, turning off the L2 prefetchers resulted in a substantial drop in throughput, highlighting their critical role for a strided access. Furthermore, our experiments indicate that the L2 prefetchers seem to track up to 32 concurrent streams on Intel's older Cascade Lake architecture and up to 72 streams on the newer Sapphire Rapids platform. Once the number of concurrent non-consecutive memory accesses exceeds this threshold, the prefetcher struggles to maintain its effectiveness, leading to reduced performance.

Take away: The parallel sequential traversal of multiple partitions can lead to significant throughput increases; however, the limits of the hardware prefetchers must be considered,

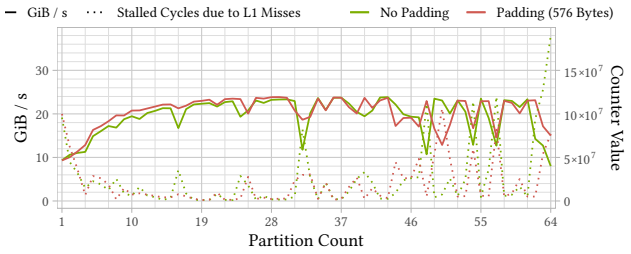


Figure 5. Impact of different paddings put between partitions to avoid that all partitions will be mapped to one cache set.

as the number of monitored streams varies between CPU architectures. In most cases, only limited information is available regarding the number of parallel monitored streams, so these parameters must be determined experimentally. The methods described in this paper can be used to determine these parameters.

Observation 3

On our systems, downward outliers in throughput were observed for specific partition counts such as 8, 16, 24, ...

Influence of cache associativity We attribute this to negative interference with the caching subsystem on the CPU that is caused by reading sequentially from multiple partitions in close temporal proximity. In our experiments, the amount of data is a power of two; therefore, certain partition counts divide the data into multiples of two, which causes the partition starting addresses to share a large number of lower bits. Since caches may use the lower parts of the address for mapping the cache line to its sets in the cache, the partitions all map their cache lines to one or only a few number of cache sets, thereby thrashing the cache even though there is still free space in the caches.

To reduce the alignment of the partitions, a 576 byte padding between the partitions was introduced. As the results in Figure 5 show, this measure is effective in reducing or even mitigating the outliers at multiples of 8, but it also introduces new outliers and retains certain old ones. The shown correlation between throughput and L1 correlated stalls substantiates the cache associativity hypothesis, though further investigation is required to better predict the performance based on partition alignment and to verify whether cache associativity is truly the sole cause of these outliers.

Take away: Data structures where multiple partitions are expected to be traversed sequentially at the same time, such as column-major tables and struct-of-array constructs, should take care to disalign the starting addresses of their partitions to a reasonable degree to avoid the negative effects of cache associativity.

4 Emerging Memory Technologies

In the previous sections, we focused exclusively on the comparison between the sequential and the strided `AggSum` variants with data residing in the local DRAM. However, modern systems are increasingly complex and offer heterogeneous memory with different latency and throughput properties. This raises the question of whether, and if so in what form, our observed insights on local DRAM can also be observed on remote DRAM – such as in classic multi-socket environments – as well as on modern emerging memory technologies like HBM or remote memory connected via CXL. Table 2 summarizes our measured peak throughput values.

Remote DRAM. In a first step, we examined traditional remote DRAM access as our Sapphire Rapids system features two sockets. For these experiments, we explicitly pinned the execution of all `AggSum` variants on a specific core on socket 1 and placed the data array in remote DRAM on socket 2. If we compare the obtained throughput values for remote DRAM with those for local DRAM (cf. Table 2), interesting observations can be made: (1) The remote DRAM throughput values for the *sequential* as well as the *simd-sequential* variant are significantly lower than for local DRAM. One reason for this will be that the access latency for remote DRAM is greater than for local DRAM. (2) In contrast, the *strided* variant shows a similar throughput for local and remote DRAM. They only differ by less than 1% in peak performance; a peak throughput of 23.96 GiB/s with 37 partitions on local DRAM and 23.78 GiB/s with 36 partitions on remote DRAM can be achieved. In this case, the memory subsystem appears to compensate very well for the higher latency with this access pattern. However, using loop unrolling, the peak throughput of local DRAM access increases by 4 GiB/s, while no significant changes for remote DRAM can be observed (cf. Table 2). Nevertheless, the results for strided access patterns are promising, since data can now be processed in both local and remote DRAM with nearly the same throughput behavior, which is not the case for sequential access patterns. This should have an effect on placement algorithms for data and functions in multi-socket environments [12, 18].

We also conducted these experiments on our Cascade Lake system consisting of four sockets. Interestingly, the older generation is more affected by accesses to remote DRAM. Although a very similar throughput curve to that of local DRAM can be observed for remote DRAM, however, we see performance losses of 30% in peak throughput for all `AggSum` variants on remote DRAM. This makes once again clear the changes to the memory subsystem on the Sapphire Rapids architecture are disruptive, as they lead to different decisions regarding an efficient data processing.

High Bandwidth Memory. In a second step, we also evaluated our different `AggSum` variants on high-bandwidth memory (HBM). On our Sapphire Rapids system, HBM is exposed as additional NUMA nodes. In addition to evaluations

Table 2. Single-threaded peak throughput results on a Sapphire Rapids system for `AggSum` over an 1 GiB array on different memory devices. We present results from only one CXL card, as both cards exhibit similar behavior.

`AggSum` variant	DRAM (local)	DRAM (remote)	HBM (local)	HBM (remote)	CXL
sequential	11.97 GiB/s	7.86 GiB/s	9.67 GiB/s	5.73 GiB/s	7.12 GiB/s
simd-sequential	19.79 GiB/s	11.88 GiB/s	18.79 GiB/s	10.02 GiB/s	10.12 GiB/s
strided	23.96 GiB/s	23.78 GiB/s	23.97 GiB/s	9.37 GiB/s	9.18 GiB/s
strided-unrolled	27.97 GiB/s	24.07 GiB/s	27.84 GiB/s	9.52 GiB/s	9.11 GiB/s

of our `AggSum` variants on local HBM, we also investigated — similar to the previous paragraph — the effects of accessing remote HBM nodes (i.e., HBM connected to a remote CPU). The obtained results are summarized in Table 2 and we can make the following observations: (1) The throughput for the *sequential* and the *simd-sequential* variants for local HBM is slightly lower than for local DRAM. This was to be expected, as the access latency for HBM is higher than for DRAM. In the most favorable case (*simd-sequential*), the difference is only 1 GiB/s. (2) As for the remote DRAM, the higher access latency can be better hidden by strided access patterns so that there are only marginal differences in throughput between local DRAM and local HBM. (3) A particularly noteworthy observation can be made when accessing remote HBM: As shown in Table 2, the throughput on remote HBM barely exceeded 10 GiB/s, whereas remote DRAM access achieved over 23 GiB/s. Even if the strided variants of `AggSum` perform better than the *sequential* variant, the *simd-sequential* variant performs best on remote HBM. The reason for this is still unknown to us.

CXL. We also evaluated our `AggSum` variants using memory expansion cards (SMART Modular CXA-4F1W; SMART Modular CXA-8F2W) connected via CXL (CXL 2.0 Type 3, i.e., CXL.mem). The system was configured such that both cards were exposed as additional NUMA nodes. Even with this novel memory type and its special interconnect, we observed improved peak performance for the strided variants compared to the sequential one — at least for a small number of partitions. However, unlike with all other memory types, this only held for the variant without SIMD optimization. In all scenarios, the highest throughput was achieved using the *simd-sequential* variant. However, these results should be currently interpreted with some caution: Even for accesses to the locally attached cards, the peak throughput remained relatively low at around 10 GiB/s, which is way below the bandwidth limits of 16 or even 8 PCIe lanes[1]

5 Related Work

The most relevant related work is the paper of Blom et al. [3]. In contrast to us, they evaluate more complex operations such as Dense Matrix Vector Multiplication, 3x3 Convolution, or Stencil Computation with different memory access patterns. They also conclude that switching from a sequential to a strided access pattern results in a performance boost. However, their comprehensive evaluation is based only on SIMDified implementations using AVX2 intrinsics without a detailed analysis of the stride size. For the SIMDified implementation of a strided access, they use the `GATHER` instruction and the efficient applicability of this instruction for such a strided access has already been also demonstrated in [7, 19]. In these papers, however, the authors clearly demonstrate that a strided access pattern provides a suitable alternative to SIMD optimization for enhancing the throughput of scalar code that employs a sequential access pattern. This aspect should be examined in more detail in future work.

[1]We are currently investigating this and will hopefully be able to present reliable results in time for the camera-ready version.

6 Conclusion and Future Work

In this paper, we presented a comprehensive experimental comparison of sequential and strided access patterns for data stored in contiguous memory on two different Intel processor generations. As we have shown, a strided access pattern with a well-chosen stride size clearly outperforms the sequential access pattern. Even a SIMD-accelerated sequential access is considerably slower than the most powerful scalar strided access, as the advanced features of the memory subsystem on the newer Intel processor generation are better utilized. In particular, higher access latencies can be effectively hidden with a strided pattern, leading to an increased throughput.

Our ongoing research activities are manifold. On the one hand, we want to broaden the evaluation scope to include AMD and ARM processors as well. In addition, we intend to conduct more detailed analyses involving novel memory technologies, as some interesting observations have to be examined in more detail. Initial results on an Apple M4 chip (ARM) indicated that similar effects can be seen (*sequential* - 30.5 GiB/s; *simd-sequential* (NEON, 128-bit) - 32.4 GiB/s; *strided-unrolled* - 59.4 GiB/s). With this broader focus, we want to better understand the strided access pattern and its influence on efficient data processing. On the other hand, we want to simplify the usage of the strided access pattern. Currently, the operations must be explicitly rewritten to adopt this access pattern as shown in Listing 1. To overcome this, it might make sense to adapt the memory allocation so that data is read sequentially at the virtual level, but the data is organized according to the strided access pattern at the physical level. In this case, the optimization would be completely transparent for applications. Therefore, the overarching goal of our ongoing research is to establish the strided access pattern as a modern way – and alternative way to SIMD – to optimize the single-thread performance.

References

[1] Daniel Abadi, Rakesh Agrawal, Anastasia Ailamaki, Magdalena Balazinska, Philip A. Bernstein, Michael J. Carey, Surajit Chaudhuri, Jeffrey Dean, AnHai Doan, Michael J. Franklin, Johannes Gehrke, Laura M. Haas, Alon Y. Halevy, Joseph M. Hellerstein, Yannis E. Ioannidis, H. V. Jagadish, Donald Kossmann, Samuel Madden, Sharad Mehrotra, Tova Milo, Jeffrey F. Naughton, Raghu Ramakrishnan, Volker Markl, Christopher Olston, Beng Chin Ooi, Christopher Ré, Dan Suciu, Michael Stonebraker, Todd Walter, and Jennifer Widom. 2016. The Beckman report on database research. *Commun. ACM* 59, 2 (2016), 92–99. doi:10.1145/2845915

[2] André Berthold, Constantin Fürst, Antonia Obersteiner, Lennart Schmidt, Dirk Habich, Wolfgang Lehner, and Horst Schirmeier. 2024. Demystifying Intel Data Streaming Accelerator for In-Memory Data Processing. In *Proceedings of the 2nd Workshop on Disruptive Memory Systems, DIMES 2024, Austin, TX, USA, 3 November 2024*. ACM, 9–16. doi:10.1145/3698783.3699383

[3] Miguel O. Blom, Kristian F. D. Rietveld, and Rob V. van Nieuwpoort. 2025. Multi-Strided Access Patterns to Boost Hardware Prefetching. In *Proceedings of the 16th ACM/SPEC International Conference on Performance Engineering, ICPE 2025, Toronto, ON, Canada, May 5-9, 2025*. ACM, 204–215. doi:10.1145/3676151.3719375

[4] Peter A. Boncz, Martin L. Kersten, and Stefan Manegold. 2008. Breaking the memory wall in MonetDB. *Commun. ACM* 51, 12 (2008), 77–85. doi:10.1145/1409360.1409380

[5] 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*. ACM, 1891–1906. doi:10.1145/2882903.2882936

[6] Andreas Geyer, Johannes Pietrzyk, Alexander Krause, Dirk Habich, Wolfgang Lehner, Christian Färber, and Thomas Willhalm. 2023. Near to Far: An Evaluation of Disaggregated Memory for In-Memory Data Processing. In *Proceedings of the 1st Workshop on Disruptive Memory Systems, DIMES 2023, Koblenz, Germany, 23 October 2023*. ACM, 16–22. doi:10.1145/3609308.3625271

[7] 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*. ACM, 9:1–9:5. doi:10.1145/3533737.3535089

[8] Tomas Karnagel, Dirk Habich, and Wolfgang Lehner. 2017. Adaptive Work Placement for Query Processing on Heterogeneous Computing Resources. *Proc. VLDB Endow.* 10, 7 (2017), 733–744. doi:10.14778/3067421.3067423

[9] Thomas Kissinger, Tim Kiefer, Benjamin Schlegel, Dirk Habich, Daniel Molka, and Wolfgang Lehner. 2014. ERIS: A NUMA-Aware In-Memory Storage Engine for Analytical Workload. In *International Workshop on Accelerating Data Management Systems Using Modern Processor and Storage Architectures - ADMS 2014, Hangzhou, China, September 1, 2014*. 74–85. http://www.adms-conf.org/2014/adms14_kissinger.pdf

[10] Roland Kühn, Jan Mühlig, and Jens Teubner. 2024. How to Be Fast and Not Furious: Looking Under the Hood of CPU Cache Prefetching. In *Proceedings of the 20th International Workshop on Data Management on New Hardware, DaMoN 2024, Santiago, Chile, 10 June 2024*, Carsten Binnig and Nesime Tatbul (Eds.). ACM, 9:1–9:10. doi:10.1145/3662010.3663451

[11] Sangjin Lee, Alberto Lerner, Philippe Bonnet, and Philippe Cudré-Mauroux. 2024. Database Kernels: Seamless Integration of Database Systems and Fast Storage via CXL. 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/p43-lee.pdf

[12] Baptiste Lepers, Vivien Quéma, and Alexandra Fedorova. 2015. Thread and Memory Placement on NUMA Systems: Asymmetry Matters. In *Proceedings of the 2015 USENIX Annual Technical Conference, USENIX ATC 2015, July 8-10, Santa Clara, CA, USA*. USENIX Association, 277–289. https://www.usenix.org/conference/atc15/technical-session/presentation/lepers

[13] Feng Li, Sudipto Das, Manoj Syamala, and Vivek R. Narasayya. 2016. Accelerating Relational Databases by Leveraging Remote Memory and RDMA. In *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*. ACM, 355–370. doi:10.1145/2882903.2882949

[14] Fabian Mahling, Marcel Weisgut, and Tilmann Rabl. 2025. Fetch Me If You Can: Evaluating CPU Cache Prefetching and Its Reliability on High Latency Memory. In *Proceedings of the 21st International Workshop on Data Management on New Hardware, DaMoN 2025, Berlin, Germany, June 22-27, 2025*. ACM, 2:1–2:9. doi:10.1145/3736227.3736231

[15] Norman May, Daniel Ritter, Andre Dossinger, Christian Faerber, and Süleyman Sirri Demirsoy. 2023. DASH: Asynchronous Hardware Data Processing Services. In *13th Conference on Innovative Data Systems Research, CIDR 2023, Amsterdam, The Netherlands, January 8-11, 2023*. www.cidrdb.org. https://www.cidrdb.org/cidr2023/papers/p6-may.pdf

[16] 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. doi:10.1007/S00778-019-00546-Z

[17] 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. doi:10.1145/2723372.2747645

[18] Iraklis Psaroudakis, Tobias Scheuer, Norman May, Abdelkader Sellami, and Anastasia Ailamaki. 2016. Adaptive NUMA-aware data placement and task scheduling for analytical workloads in main-memory column-stores. *Proc. VLDB Endow.* 10, 2 (2016), 37–48. doi:10.14778/3015274.3015275

[19] Lennart Schmidt, Johannes Pietrzyk, Juliana Hildebrandt, Alexander Krause, Dirk Habich, and Wolfgang Lehner. 2025. Rethinking MIMD-SIMD Interplay for Analytical Query Processing in In-Memory Database Engines. In *15th Conference on Innovative Data Systems Research, CIDR 2023, Amsterdam, The Netherlands, January 19-22, 2025*. www.cidrdb.org. https://www.vldb.org/cidrdb/papers/2025/p7-schmidt.pdf

[20] Xinjun Yang, Yingqiang Zhang, Hao Chen, Feifei Li, Gerry Fan, Yang Kong, Bo Wang, Jing Fang, Yuhui Wang, Tao Huang, Wenpu Hu, Jim Kao, and Jianping Jiang. 2025. Unlocking the Potential of CXL for Disaggregated Memory in Cloud-Native Databases. In *Companion of the 2025 International Conference on Management of Data, SIGMOD/PODS 2025, Berlin, Germany, June 22-27, 2025*. ACM, 689–702. doi:10.1145/3722212.3724460