SeaCache: Efficient and Adaptive Caching for Sparse Accelerators

Xintong Li
Tsinghua University
Beijing, China
lixt21@mails.tsinghua.edu.cn

Jinchen Jiang
Tsinghua University
Beijing, China
jiangjc22@mails.tsinghua.edu.cn

Mingyu Gao
Tsinghua University
Beijing, China
Shanghai Qi Zhi Institute
Shanghai, China
gaomy@tsinghua.edu.cn

Abstract

Sparse tensor computations are highly memory-bound, making on-chip data reuse in SRAM buffers critical to the performance of domain-specific sparse accelerators. On-demand caches are commonly used in recent sparse accelerators, due to the advantage of easy space allocation and the ability to avoid unnecessary data fetches compared to scratchpad-style data buffering. However, existing cache designs suffer from inefficient space utilization due to the difficulty of fitting variable-length sparse data into fixed-size cache blocks. The theoretically optimal replacement policies used by them also have substantial implementation cost, requiring significant on-chip space to manage the metadata. To address these issues, we propose SeaCache to enable efficient and adaptive caching for sparse accelerators. SeaCache includes three key techniques. First, it incorporates fiber packing and splitting to efficiently map variable-length data into fixed-size cache blocks with high space utilization. Second, it proposes a practical replacement policy that performs similarly to the optimal one but has a much cheaper implementation. Third, it shares the cache space between the actual data and the replacement policy metadata, with a two-phase adaptive mechanism to decide the best partition ratio. Overall, SeaCache outperforms state-of-the-art sparse cache designs by 2.8× on average, demonstrating the effectiveness of its novel optimizations.

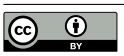
CCS Concepts

• **Computer systems organization** → **Special purpose systems**; • **Computing methodologies** → **Linear algebra algorithms**; • **Theory of computation** → Caching and paging algorithms.

Keywords

sparse tensor algebra, hardware acceleration, cache, replacement policy

ACM Reference Format:
Xintong Li, Jinchen Jiang, and Mingyu Gao. 2025. SeaCache: Efficient and Adaptive Caching for Sparse Accelerators. In *58th IEEE/ACM International Symposium on Microarchitecture (MICRO '25), October 18–22, 2025, Seoul, Republic of Korea.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3725843.3756040



MICRO '25, Seoul, Republic of Korea

ACM ISBN 979-8-4007-1573-0/25/10
https://doi.org/10.1145/3725843.3756040

1 Introduction

Sparse tensors play a crucial role in various fields, such as graph analysis, high-performance computing, and machine learning. However, their irregular data patterns often lead to inefficient computations on conventional general-purpose processors, mainly due to random memory accesses to sparse data and uneven workload distribution across parallel processing units. To mitigate these issues, specialized accelerators have been developed to optimize key sparse tensor operations, like sparse-sparse matrix multiplication. These accelerators feature specialized arrays of multiply-accumulate computing units and on-chip large-capacity SRAM hierarchies. To overcome the memory access bottleneck in sparse tensor computations, various dataflow optimizations are further applied, such as reordering the nested loops to balance data reuse with Inner Product [13, 25], Outer Product [14, 22, 38], or Gustavson's schemes [3, 15, 18, 37], and tiling the loops either statically [17, 35] or dynamically [16, 21] to allow local subsets of tensor data to fit in the limited on-chip SRAM.

Despite the extensive collection of research on the optimizations of sparse accelerators, we find that the specific design of the on-chip SRAM itself, which is the most critical component to realize data reuse, has not been sufficiently explored. Different from regular dense computations, the *scratchpad-style, explicit decoupled data buffering* [24] does not work well on sparse tensor operations. The variable and irregular sizes of sparse data make buffer space allocation challenging, and simply and blindly bringing all sparse data to the buffer may result in unnecessary accesses if some data are never needed. As a result, many recent designs start to use *on-demand data caching* for sparse accelerators [3, 18, 22, 37, 38].

In this paper, we focus on two aspects of cache design for sparse accelerators: data mapping schemes and replacement policies. For cache data mapping, previous work [30] has proposed to directly use the IDs of sparse rows/columns (called fibers [32]) to index the cache, avoiding the translation to physical addresses and thus saving latency and metadata accesses. However, the drastically varying sizes of sparse fibers make them difficult to fit in the fixed-size cache blocks, resulting in cache space underutilization and excessive cache misses. For replacement policies, existing designs have noticed that the structural information of one operand tensor can guide the replacement decisions of the other tensor [3, 4, 38], leading to the guided LRU (gLRU) policy that realizes the Belady's Optimal [6]. Nevertheless, gLRU has a substantial implementation cost, which either doubles metadata accesses to the off-chip memory, or requires significant on-chip space to dynamically reorganize the guide metadata.

To address these issues, we propose SeaCache, an efficient and adaptive cache design for sparse accelerators. SeaCache only modifies the on-chip SRAM components, is compatible with banking, and is orthogonal to the processing element design of the accelerator. SeaCache consists of three key novel techniques that improve the cache mapping scheme and the replacement policy.

First, to efficiently map variable-length sparse fibers into fixed-size cache blocks, SeaCache incorporates *fiber packing and splitting*. Each cache block may pack multiple short fibers, or keep a fiber segment that is split from a long fiber. These techniques improve cache space utilization for short fibers, and allow more data cached from long fibers to reduce cache misses. We design the corresponding mapping and format schemes to support such flexible packing and splitting, with only small area overheads on the cache array.

Second, we move away from the expensive gLRU to a *guided LFU* (gLFU) replacement policy that is more practical to implement. Instead of keeping complex recency lists that involve many wide pointers in gLRU, gLFU only requires simple narrow counters, thus greatly saving the metadata size. While gLFU is no longer provably optimal as gLRU, its empirical performance is similar. We introduce additional shadow ways without data blocks in each cache set, called *virtual tags*, to accommodate more gLFU counters on-chip that capture future access patterns and improve replacement accuracy. We also add one more port to the cache tag array to prevent counter updates from interfering with normal accesses.

Third, deriving the gLFU counter values requires future access information, which follows the structural metadata of the other operand tensor. Thus we need to prefetch a future window of data from the latter tensor. This requires on-chip storage space. We let these prefetched data share the cache space with the actual data, and propose a *two-phase adaptive* mechanism to select the appropriate size to allocate to the prefetched data. In the offline phase, we estimate an initial prefetch size based on simple sparsity statistics of the tensor. During the online phase, we collect runtime performance metrics and use them to dynamically adjust the prefetch size.

We demonstrate the effectiveness of SeaCache by integrating the three techniques in a sparse accelerator that uses the Gustavson's dataflow and supports statically determined tiling. When compared with state-of-the-art cache designs in sparse accelerators [3, 30, 38], SeaCache achieves a 2.8× speedup on average, by significantly reducing the cache miss rate and the memory bandwidth usage with the proposed optimizations. When enabled on top of a base design, fiber packing and splitting, guided LFU with virtual tags, and two-phase adaptive prefetch size selection contribute 1.80×, 1.10×, and 1.38× improvements, respectively. We also show that SeaCache outperforms a highly optimized scratchpad-style design [35] by 2.1×, demonstrating the high potential of on-demand caching for sparse accelerators.

We make the following contributions in this paper.

- We propose the fiber packing and splitting strategy to map variable-length sparse fibers into fixed-size cache blocks for sparse accelerators, with high space utilization.
- We propose the guided LFU policy as a practical and near-optimal replacement policy for sparse accelerators, and the corresponding hardware design that incorporates virtual tags to more accurately capture future access patterns.
- We propose the two-phase adaptive mechanism to determine the best cache partition ratio between the prefetched metadata used by the guided LFU policy and the actual data.
- We integrate the above three techniques in a sparse accelerator and demonstrate a 2.8× speedup on average over state-of-the-art designs.

2 Background

2.1 Sparse Tensor Algebra

Sparse tensors are essentially multi-dimensional arrays with most elements being 0. In this paper, we follow the common terminology [32], in which a tensor element is called a *point* at a certain *coordinate*, e.g., $X_{i,j,k}$ at (i, j, k). To save storage space and computation, sparse tensors are often represented in various *compressed formats*, such as the coordinate format (COO), compressed sparse row/column (CSR/CSC), and block variants like block CSR. These formats follow a common structure that organizes tensor dimensions into a hierarchy of *fibers* [32]. Each fiber represents a sequential list of coordinates and their corresponding non-zero values along the specific dimension. The term *position* refers to the actual storage location of a point within these compressed formats, which typically differs from its coordinate. We further use the Einsum notation [10] to represent operations on sparse tensors, such as the sparse-sparse matrix multiplication (SpMSpM) between tensors A $(I \times K)$ and B $(K \times J)$, which is $C_{i,j} = A_{i,k} \times B_{k,j}$ over (I, J, K).

2.2 Sparse Accelerators and Sparse Dataflow

Many specialized hardware accelerators have been developed for sparse tensor operations [8, 12, 19, 23, 26, 34, 36]. Due to the sparsity, each data element is involved in computations with only the non-zero elements of other operand tensors, resulting in less data reuse than the dense scenarios. Consequently, sparse accelerators are usually heavily memory-bound. Loop reordering and loop tiling techniques, which have been demonstrated to be efficient in dense computations, have recently been adopted for sparse accelerators.

Loop reordering. Consider the SpMSpM $C_{i,j} = A_{i,k} \times B_{k,j}$ with three dimensions (i, j, k). By reordering the dimensions in the loop nest, we have three main schemes of sparse dataflow, namely Inner Product (IP) [13, 25], Outer Product (OP) [14, 22, 38], and Gustavson's (Gust) [3, 15, 18, 37]. These different loop orders affect the reuse of the three tensors. For example, the k loop resides at the innermost level in IP, which maximally reuses the output C, while it is the outermost one in OP, which has poor output data reuse. There also exist designs supporting dynamic reconfiguration between multiple dataflow schemes to adapt to various sparse patterns [18, 20]. We mainly use Gust in this paper, as it achieves a relatively balanced and efficient design point. We discuss other dataflows in Section 4.4. In Gust, each $A_{i,k}$ element in the ith row of A multiplies with the corresponding kth row of B, and the partial product is accumulated into the corresponding ith row of C; i.e., $C_i = \sum_k A_{i,k} \times B_k$.

Loop tiling. Another approach to improving data reuse is to split large tensors along some loop dimensions into smaller *tiles*, which can better fit in the limited on-chip SRAM. Tiling is usually done on the coordinate space rather than the position space, in order to ensure matched coordinate spans between tensor tiles

during computation. However, due to varying data sparsity, simple static and fixed-size tiling would lead to diverse tile sizes that either underutilize or exceed the SRAM capacity. Therefore, state-of-the-art designs leverage data pre-sampling to better choose the proper tile size [35], use fully dynamic approaches to better adapt to the data sparsity [16, 21], or combine static and dynamic methods [17]. For simplicity, in this work, we determine the optimized tile size with offline pre-sampling, but the dynamic tiling methods can also be integrated with our design (see Section 4.4).

2.3 Cache Management for Sparse Accelerators

The memory-bound nature of sparse accelerators makes them heavily rely on efficient on-chip data reuse. Explicit decoupled scratchpad buffering [24] and on-demand caching are two common approaches used in hardware accelerators. We note that while scratchpads are more efficient for regular dense computations, caches are potentially a better choice for sparse accelerators. There are several reasons. First, the main data structures, sparse fibers, have irregular and diverse sizes, which complicates space allocation and would cause fragmentation in the scratchpad. It would be easier to apply a fixed-size block granularity for space management with sparse data. Second, because of the sparsity, not all fibers are needed by the computations. For example, in the Gust dataflow, if all the $A_{i,k}$ elements in column k are zero, the B_k row will never be used and thus there is no need to fetch it on-chip. These savings are easier to realize with on-demand caches than with scratchpads, especially with the proper replacement policies described below. Third, typical caching overheads, such as translation between row/column IDs and physical addresses, on-demand data fetch latencies, and tag storage overheads, have been greatly alleviated with recent design advances, as described below. As a result, caches have become the common choice in many previous sparse accelerator designs [3, 18, 22, 37, 38]. In this work, we focus on caches and aim to optimize two major aspects in cache design: data mapping schemes and replacement policies. We will show that our optimized cache design outperforms the scratchpad in Section 6.

2.3.1 Mapping Schemes. Conventional caches typically use the physical address to determine where to map a block of data in the cache. In sparse accelerators, the data elements inside a fiber (e.g., a row or a column) are often fetched and used together. This provides an opportunity to directly use the fiber IDs to index the cache, as proposed in the X-Cache design [30]. This approach eliminates translation between the fiber IDs and the physical addresses, which would require frequent accesses to the metadata of the compressed format, such as the row pointers in the CSR format. X-Cache has been shown to reduce the load-to-use latency by approximately 40% in SpMSpM [30]. In addition, now we do not need to access most metadata of the compressed format. For example, in CSR, when a certain row is needed, we can use its ID to directly index the cache, without first obtaining the row pointers to calculate the positions of non-zero column IDs and values.

Figure 1 shows a simple example about how the fiber-ID-based mapping works in X-Cache. Assume that the cache for tensor B has two blocks, each with the size of two elements. In the Gust dataflow, the elements a and b in A need rows 0 and 3 of B, which are fetched and stored in the cache (left bottom of the figure). Instead

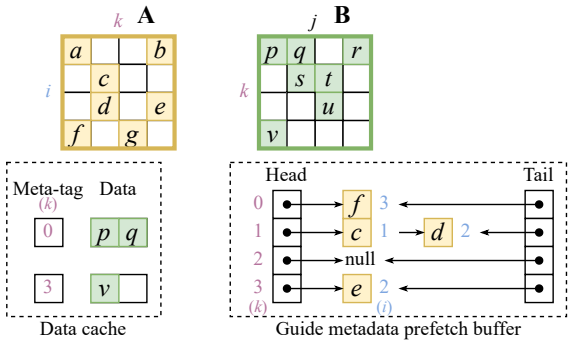


Figure 1: An example to illustrate how the fiber-ID-based mapping scheme and the guided LRU policy work for SpM-SpM between A and B.

of the physical addresses, the cache keeps the meta-tags, which are essentially the row IDs, 0 and 3, for the two blocks.

However, a critical challenge arises from the drastically varying lengths of the fibers in the sparse tensors. With a fixed cache block size, if the fiber is shorter than the block size, some cache space would be underutilized, as in the block in row 3. If the fiber is too long, it cannot fit in the block. The original X-Cache paper did not explicitly describe how to handle this case. In this paper, we assume a reasonable design in which only the first few elements are stored in the block while the rest are discarded (as in the block of row 0 in the figure), following the overflow handling method in Tailors [35]. Such partial caching is a better choice than streaming in these thrashing cases, as it retains reuse at least for some data.

2.3.2 Replacement Policies. Recall that the theoretical optimal replacement policy, Belady's Optimal [6], suggests replacing the cache block with the next farthest access time. While in general-purpose applications such future information is unknown, in sparse computations like SpMSpM, the access pattern of one tensor is determined by the non-zero element distribution of the other tensor. For example, in the Gust dataflow, which B rows are needed depends on the column ID k of the $A_{i,k}$ element. Thus, the structural metadata of A can be leveraged to guide the replacement of B data.

This replacement policy, which we call *guided LRU* (gLRU), has been exploited by several previous designs, including P-OPT [4], InnerSP [3], and SpArch [38]. As shown in Figure 1, assume that we have just finished multiplying element $A_{0,0} = a$ with row B_0 and $A_{0,3} = b$ with B_3, and are about to multiply $A_{1,1} = c$ with B_1. The cache has already been fully filled with two rows of B_0 and B_3, and one of them needs to be evicted to make room for the incoming B_1. Looking at the non-zero distribution of tensor A, we find that the next time to reuse B_0 is for the next non-zero element $A_{3,0} = f$ in column $k = 0$, while that for B_3 is for $A_{2,3} = e$. Hence B_3 has a more recent reuse at $i = 2$ than B_0 at $i = 3$, and we should evict B_0. Essentially, by looking at each column k of A and identifying the next non-zero row ID i, we can know when the corresponding B_k row would be reused and make optimal replacement decisions.

Note that achieving gLRU requires the transposed CSC format of A in addition to the original CSR format for computations, in order to examine each column of A. P-OPT [4] and InnerSP [3]

Table 1: Comparison of cache management techniques between state-of-the-art designs and ours.

Design	Block size	Replacement policy
P-OPT [4]	64 bytes	gLRU (offline transpose)
InnerSP [3]	64 bytes	gLRU (offline transpose)
SpArch [38]	576 bytes	gLRU (1/6 prefetch size)
X-Cache [30]	16 bytes	LRU
SeaCache (**ours**)	Fiber packing & splitting in 64 bytes	gLFU (adaptive prefetch size shared with cache)

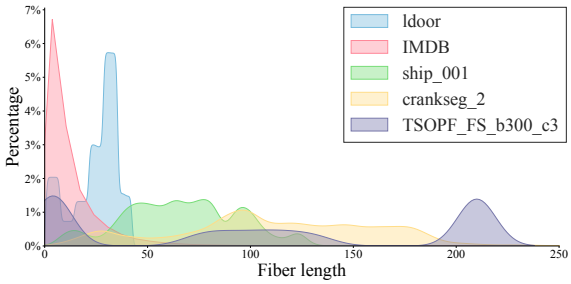


Figure 2: Distributions of row fiber non-zero lengths in selected sparse matrices from SuiteSparse [9].

pre-computed such a transposition and assumed that both formats were available for access. This is only possible if the tensor remains constant and can be reused multiple times to amortize the offline transpose cost. SpArch [38] instead claimed to perform next-reuse detection online, but did not provide microarchitectural design details. In this work, we assume an efficient implementation as illustrated in Figure 1. The hardware would prefetch a certain number of A elements into a guide metadata buffer. These metadata are reorganized by column ID k, into linked lists, each ordered by row ID i. Consequently, the row ID of the head element in the list of column k represents the next reuse time for the B_k row. As in Figure 1, the prefetch size is 4 elements, $\{c, d, e, f\}$, following its original CSR order and reorganized into columns on-chip. From their linked list structures, we see that the next reuse time for each B_k would be $3, 1, \infty, 2$. Thus, B_2 has the farthest reuse and is the best candidate for replacement. When an element in the prefetch buffer, e.g., c, finishes its computation, it is removed from the list head. Simultaneously, we prefetch a new element, e.g., g, and append it to the corresponding list tail. This follows a sliding window manner.

3 Motivation

In this section, we focus on cache management for sparse accelerators, and discuss several key problems in the state-of-the-art designs introduced in Section 2.3, mainly concentrating on the aspects of mapping schemes and replacement policies. The comparison is summarized in Table 1.

3.1 Mapping of Variable-Length Fibers

As already discussed in Section 2.3.1, when the fiber length mismatches with the cache block size, either the cache space would be underutilized for short fibers, or there will be many cache misses for uncached elements in long fibers. Indeed, we find that for real-world sparse matrices (obtained from the SuiteSparse Matrix Collection [9]), not only the fiber lengths in different matrices, but also those within a single matrix, vary significantly as shown in Figure 2. This makes it difficult to choose a single fixed optimal block size.

Previous sparse accelerators only use fixed block sizes that fit their target applications, but cannot generalize to diverse sparse patterns. For example, X-Cache [30] opts for very short block sizes of 1 or 2 elements. This works well in scenarios with very high sparsity and mostly short fibers, increasing cache space utilization and reducing internal fragmentation. However, long fibers cannot fit in these small blocks and many elements remain uncached. On the other hand, SpArch [38] uses a longer cache block size of 48 elements, which achieves higher throughput and lower latency by avoiding multiple cache probes when accessing long fibers. This design performs better in more mildly sparse scenarios. But such large blocks will be underutilized for short fibers.

Design goal 1: To better support the highly diverse fiber lengths, we need a more space-efficient mapping scheme to handle fibers both shorter and longer than the block size, ensuring good performance across various scenarios.

3.2 Cost of Guided Replacement

For replacement policies, although gLRU achieves optimal eviction decisions, its implementation is non-trivial and may cause substantial overheads. The offline approaches in P-OPT and InnerSP [3, 4] allow for simpler hardware designs but introduce additional cost for offline transposition and extra accesses to the transposed matrix. Note that transposing a sparse matrix in a compressed format has substantial cost, up to 126% compared to the actual computation even with highly optimized and dedicated hardware support [11]. Although repeated computations on the same matrix may amortize this cost, it is unacceptable in scenarios where the matrix is computed only once. Furthermore, the offline methods still incur extra online overheads because of the need to access the separate transposed matrix, i.e., A^T. This cost of doubling the accesses to A would become especially significant considering that the other matrix B now enjoys good reuse with the optimal gLRU policy.

On the other hand, the online approach described in Section 2.3.2 (pre)fetches A only once and uses its data for both guided replacement decisions and actual computations. However, it has its own drawbacks. First, significant on-chip SRAM space is needed to store the prefetched guide metadata of A. To obtain accurate future reuse information, the prefetch size needs to be large enough, so that a sufficient number of columns k can identify at least one non-zero element to make the linked lists non-empty (c.f. Figure 1). This is particularly necessary when the tensor is very sparse with few non-zeros. Second, complex hardware logic and extra auxiliary data are needed to support online transposition. Recall Section 2.3.2 and Figure 1 that each column k needs a head pointer and a tail pointer, both to the prefetched metadata of A. Additionally, each element in the prefetch buffer also needs a pointer to the next one in the list. The number of head/tail pointers is linear to the dimension size K (or the tiled size T_K if with tiling), which can be much larger than the number of non-zero elements. Furthermore, the bitwidth of

each pointer is also not small, requiring ~ 20 bits to cover a typical prefetch buffer in hundreds of kB to a few MB.

Design goal 2: We aim to develop a guided replacement policy that achieves similar efficiency, but is simpler and requires less implementation cost than gLRU.

One known approach to alleviating the extra on-chip storage for the guide metadata is to share the existing data cache space, as proposed by P-OPT [4]. However, the offline method of P-OPT pins the entire guide metadata on-chip and significantly reduces the available space for data caching. The online approach only needs to prefetch a subset of metadata ahead, but it faces the question of deciding this prefetch size. A small prefetch size degrades the accuracy of future reuse information, and hence decreases the hit rate. A too large prefetch size would take too much space from the data cache, also reducing the cache hits. Intuitively, the necessary guide metadata to prefetch depend on the sparse pattern of the tensor. A denser tensor can more quickly identify the next non-zero coordinate for the next reuse even with a small prefetch size, while a sparser tensor may need to look farther more into the future.

Design goal 3: When sharing the on-chip SRAM space between the actual data and the guide metadata, we need an adaptive mechanism to decide the partitioning ratio between them to achieve the overall best data hit rate.

4 SeaCache Design

We propose SeaCache, a set of efficient and adaptive cache management techniques for sparse accelerators. SeaCache mainly improves the mapping schemes and the replacement policies. It includes *fiber packing and splitting* techniques that efficiently map variable-length fibers to fixed-size cache blocks (Section 4.1), a practical *guided LFU* replacement policy that is easier to implement than guided LRU (Section 4.2), and a *two-phase adaptive* mechanism to select the prefetch size of the guide metadata (Section 4.3). The above three techniques realize the three design goals in Section 3. We discuss how to integrate them together in Section 4.4 at the end.

4.1 Fiber Packing and Splitting

To support highly variable fiber lengths (**design goal 1**), several architectural solutions are possible. One straightforward way is to deploy multiple SRAM banks with different block sizes that respectively fit long and short fibers. However, the ratio of long to short fibers in a tensor is not known a priori and may vary across different local regions, making the static bank-level separation underutilize the cache resources. Another solution is to use reconfigurable block sizes that allow banks to be configured in different modes for long and short fibers. Such reconfiguration needs to happen dynamically to adapt to the ratio of long and short fibers. However, the long and short modes with different block sizes need to use different address indexing schemes to map fibers to SRAM locations. Such mode switching requires a full remapping of the entire bank, i.e., flushing and refetching all the data, which incurs excessive overheads.

We propose an efficient scheme that uses fixed-size hardware blocks, but flexibly maps variable-length fibers with high space utilization, by *packing* multiple short fibers into one block, or *splitting* a long fiber into multiple segments to store across multiple blocks. Figure 3 shows our mapping scheme, with a set-associative cache

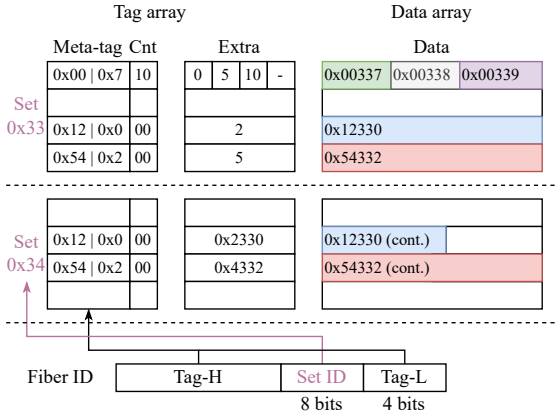


Figure 3: Tag and data array structures for the mapping scheme with fiber packing and splitting.

organization. In this example, we use a block size of 64 bytes. With fiber packing and splitting, each block may store (a segment of) a long fiber or multiple short fibers. To indicate the number of fibers in a block, two extra `Cnt` bits per block are added to the tag part. A value of `00` represents that only one fiber or a fiber segment is stored. Otherwise, there are `Cnt`+1 fibers in this block. We also add 16 `Extra` metadata bits per block in the tag array, whose usage will be discussed below. Because the `Extra` bits are used in different ways for fiber packing and splitting, each block must be in one of the two modes (distinguished by `Cnt`), but not both.

Fiber packing. When multiple fibers reside in the same block, they must have the same set ID to legally be placed together. To save space for tag bits, we further require that they have contiguous fiber IDs, so we only need to store the tag of the first ID in the tag bits. In other words, the `Cnt`+1 fibers in the block have their tags as `Tag`, `Tag`+1, . . . , `Tag`+`Cnt`. This would require us to slightly change how fiber IDs are mapped to sets. In normal fiber-ID-based mapping schemes, the set ID is from the least significant bits of the fiber ID, so we spread consecutive fibers into different sets (note that there are no offset bits). To be able to pack multiple consecutive fibers in one block, we skip a few least significant bits when taking the set ID, shown as the `Tag-L` bits at the bottom of Figure 3. The `Tag-L` and `Tag-H` bits are concatenated as the full tag. For example, the first block in Figure 3 contains three fibers (`Cnt` = 10) with IDs of `0x00337`, `0x00338`, `0x00339`, whose tags are contiguous as `0x00|7`, `0x00|8`, `0x00|9`. While the 2-bit `Cnt` means at most four fibers can be packed in one block, we choose a slightly wider `Tag-L` of 4 bits to support unaligned cases, e.g., when fiber IDs of 7, 8, 9 reside in one block as above. More analysis on `Tag-L` is done in Section 6.4.

To locate the multiple fibers in a block, we use the 16 `Extra` bits, encoded as four 4-bit offsets within the block. In the above block, the three fibers are at positions 0, 5, 10, respectively.

Fiber splitting. On the other hand, when a long fiber is split and stored in multiple blocks, we need to adjust the indexing of each segment. Otherwise, all segments would have exactly the same set ID and tag from the same fiber ID and would be restricted in a single block as in previous designs [30, 38]. Specifically, for the lth

segment, we map it using an adjusted fiber ID of $\texttt{ID} + (l \ll n_{\texttt{Tag-L}})$, where the added term changes the set ID bits, so each segment is mapped to a different set. To differentiate a segment in this adjusted form from the fiber with the same ID, we further store the lowest 16 bits of the original fiber ID in the `Extra` bits. These bits are checked when the block is accessed to ensure the correct match. For the first segment which does not use an adjusted ID, we do not need to store the original fiber ID, so we put the number of segments of the fiber in the `Extra` bits.

For example, fiber `0x12330` is longer than a block and split into two segments. The first segment is mapped to the set `0x33` in the normal way, with the `Extra` field being 2, representing that the fiber has 2 segments. The second segment with $l = 1$ has an adjusted ID of `0x12340` following the above equation and is thus mapped to the set `0x34`. The `Extra` bits are `0x2330`, extracted from its original fiber ID, so we will not mismatch it with the actual fiber `0x12340` even though their tag bits are the same. That is, only one of the two can appear in the cache.

Using only the lowest 16 bits to differentiate fibers poses a limit on the number of segments allowed. Assume two fibers with IDs of a and b, $a \neq b$. When their l_a-th and l_b-th segments are mapped to the same set and have the same tag, we have $a+(l_a \ll 4) = b+(l_b \ll 4)$. If their `Extra` bits are also identical, we have $a = b \pmod{65536}$. To satisfy all the above equations, we must have l_a or $l_b \geq 4096$ blocks, or 256 kB. With each element containing a 64-bit value and a 32-bit coordinate, this translates to 21854 elements (21845 = 256 kB / (32 bit + 64 bit)) in a fiber. Almost all sparse matrices in SuiteSparse are within this limit. Furthermore, when tiling is applied, a single long row/column would be split into multiple shorter fibers. In the rare cases of very long fibers, we can reduce the tile size, or simply discard the remaining elements beyond the allowed segments.

Cache access process. Following the fiber-ID-based mapping approach [30], to access a fiber in the cache, we use its ID to directly index the cache. We extract the set ID bits from the fiber ID to identify the target set, and concatenate `Tag-H` and `Tag-L` to compare with the tags in this set (Figure 3 bottom). Such tag matching is performed in a fuzzy way for the lowest few bits, so as long as the request tag falls in the range [`Tag`, `Tag + Cnt`] for a way in the set, it is a potential hit. If `Cnt` is larger than 00, the block is in the packing mode. We decode the `Extra` bits into four offsets, and follow the corresponding one to access the data. If `Cnt` is 00, the block is in the splitting mode. We read the `Extra` bits to get how many segments to access. For each segment, we calculate the adjusted ID to determine the corresponding set and access the data from that set. For segments other than the first one, we also check the `Extra` bits to match the lowest bits of the original fiber ID.

In the above cache hit case, no translation to physical addresses is performed. However, for a miss, we need to follow the conventional sparse data access flow, to retrieve the metadata from the memory first, translate to the physical address, and then access the data from the memory.

SRAM capacity cost. The additional bits added to each block in our new mapping scheme introduce minor area overheads. The original tag field is usually ~ 20 bits with 32-bit coordinates and ~ 10 bits for set IDs. With fiber packing, the data block size can be much larger without worrying about space waste. We find that a size of 64 bytes is a balanced choice (Section 6.4). In addition to these fields, we add 2 `Cnt` bits and 16 `Extra` bits, in total 18 bits. These are only $18/(20 + 64 \times 8) = 3.4\%$ overheads.

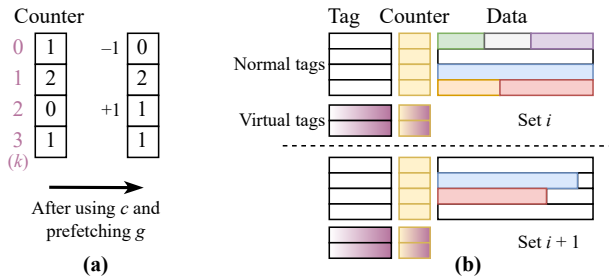


Figure 4: The guided LFU replacement policy. (a) The per-column reuse counters and their updates. (b) The counters are integrated into the data cache, with additional virtual tags in each set.

Comparison to compressed caches. The fiber packing technique shares similar ideas to prior compressed cache designs [2, 27–29] that compress multiple memory blocks into one cache block to increase the effective cache capacity. However, SeaCache has two main differences. First, in the domain of sparse tensors, both data distributions and access patterns are more predictable than general-purpose workloads. The cached A and B fibers are read-only, and have fixed (albeit diverse) lengths. Thus we can avoid complicated (de)compression and not deal with compress ratio changes. Second, compressed caches only pack blocks but do not split. Fiber splitting is unique to the fiber-ID-based mapping approach [30, 38].

4.2 Guided LFU Replacement

As discussed in Section 3.2, while gLRU is optimal, its implementation cost is excessive due to the many pointers to maintain the per-column LRU linked lists. To avoid such overheads (**design goal 2**), we propose to move from LRU to LFU, which is also a commonly used and empirically well-performed replacement policy. Specifically, we use a guided LFU (gLFU) policy, where we use the future reuse information from matrix A to derive the reuse counts (a.k.a., frequencies) of each fiber of B, instead of the reuse distance in gLRU. Consequently, the per-column linked list becomes a single counter per column. Figure 4(a) shows the auxiliary metadata needed by gLFU, in contrast to those in Figure 1. When the prefetch buffer slides to the next element, i.e., when $A_{1,1} = c$ is used and $A_{3,2} = g$ is prefetched, their corresponding column counters are decremented and incremented, respectively.

The metadata savings from gLRU to gLFU can be significant. Recall that each pointer in the gLRU lists can be as wide as 20 bits. In contrast, the counters in gLFU do not need to be very wide because of the sparsity of tensors; each column usually only has a few non-zeros within the prefetch size. In our design, we find 4-bit counters are sufficient. Even if counter overflow (i.e., saturation) occurs, the error is tolerable because these fibers are already marked as important with very large counts and are unlikely to be evicted. Therefore, besides the prefetched structural metadata of A, gLRU needs additional $20 \times (2K + S)$ bits, while gLFU only needs $4K$ bits.

The tradeoff here is that gLFU is no longer provably optimal as gLRU. However, empirically we find gLFU performs as good as

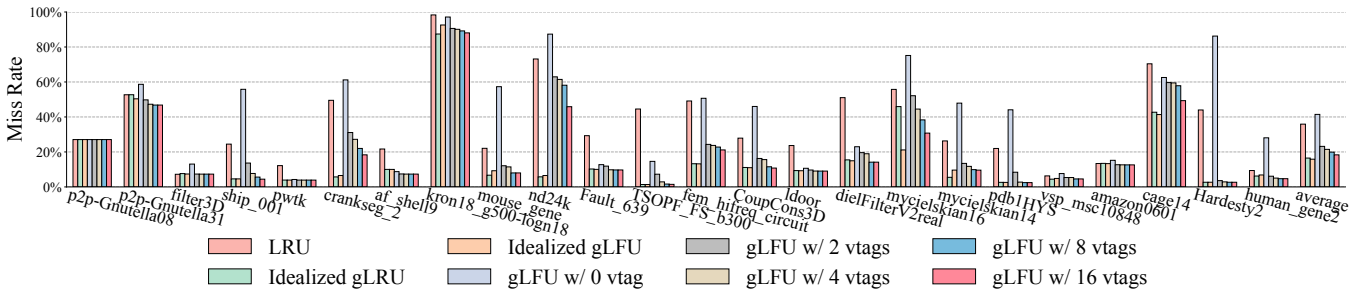


Figure 5: Comparison of miss rates with different replacement policies, including LRU, idealized gLRU, idealized gLFU, and gLFU with various numbers of virtual tags per set.

gLRU. The first three bars of each group in Figure 5 compare (idealized) gLRU and gLFU against the conventional LRU policy. These idealized guided policies have sufficiently large on-chip SRAM to keep their metadata as calculated in the previous paragraph. We see that both guided policies significantly reduce the miss rates over LRU, with 16.5% and 15.9% miss rates, respectively. Note that, here, gLFU even outperforms gLRU, because gLRU cannot achieve Belady's Optimal with a limited prefetch size (see below).

Practical implementation. However, the $4K$ bits of counters may still be expensive if dedicated SRAM is used to store them on-chip. In particular, for highly sparse matrices such as `web-Google` and `kkt_power`, the SRAM capacity for these counters may even exceed the structural metadata of A, as the number of columns K could be larger than the number of non-zeros in the prefetch buffer.

We leverage two key insights here. First, the gLFU counters can be *sparse* for highly sparse tensors; i.e., many counters are zero and do not need to be stored. Second, most of the counters that are important to making replacement decisions are for those fibers that are *already cached* in the data cache. This is because these fibers are exactly the replacement candidates. Therefore, we propose to integrate the counters into the data cache tag array, exactly like how the conventional LRU bits are maintained per block besides the tags in standard caches. Only the cached fibers have their counters valid. This integrated design has several benefits. First, compared to using a separate counter cache, we reuse the same tag space for both counters and data, avoiding duplication. Second, when an A element is actually used and moved out from the head of the prefetch buffer (e.g., c in the previous example), the decrement of the corresponding counter can be piggybacked on the access to the fiber data, within a single cache access.

However, we still have two problems. First, as we aim to analyze future access patterns with gLFU, some uncached fibers that will be accessed in the future are still important, but their counters are missing, which may significantly degrade performance compared to the idealized case. This can be seen in Figure 5, where the miss rates of "gLFU w/ 0 vtag" noticeably increase by 2.6× over "idealized gLFU". To resolve this issue, we further add several *virtual tags* in each cache set, which only have the counters without actual fiber data, as shown in Figure 4(b). These virtual tags can store counters for uncached fibers that will be accessed in the future. Only a moderate number of virtual tags are needed, such as 2 to 4 virtual tags besides the 16 ways in a set as indicated by Figure 5. We use 4 virtual tags per set. This adds about 20 bits × 4 ways + 4 bits × (16 ways + 4 ways) = 160 bits in each set. Combined with the extra bits added in Section 4.1, the total overheads are about 5.3% more bits.

Second, maintaining the counters requires extra cache probes. We have already discussed how to merge counter decrements with actual data accesses. But counter increments when prefetching a new A element (e.g., g in the previous example) still need separate cache probes. Fortunately, these extra probes only access the tag array but not the data array. We add an extra port to the tag array in hardware to support two accesses per cycle, with moderate area cost (see Table 2). Note that such doubled (or even tripled) cache probes are not unique to our gLFU policy. Any guided policy would require such metadata maintenance.

Overall flow. We finally describe the overall workflow of gLFU with virtual tags. The gLFU counters need to be incremented and decremented when an A element is moved into and out of the prefetch buffer, respectively. If the counter exists in the cache, either with a normal tag or a virtual tag, it is directly updated. If the counter is not present, we assume an initial counter value of zero. If it is incremented to 1 (in case of prefetching), we check whether the counter of a virtual tag is smaller (i.e., invalid or 0). If so, we replace it with the new counter. Here we only check virtual tags because for this prefetching, the actual B fiber data do not need to be fetched at this time, so the data space of a normal way would not be used.

4.3 Selection of Prefetch Size

Regardless of whether gLRU or gLFU is used, part of matrix A needs to be prefetched onto the chip to guide replacement decisions. We follow the P-OPT design [4] to directly store the prefetched data in the data cache, i.e., sharing the cache space between the actual B fiber data and the prefetched A data. This avoids the need for a separate dedicated prefetch buffer. However, now we need to carefully determine how to partition the cache space between the two types of data (**design goal 3**).

We find that the best prefetch size could vary significantly between different tensors. Figure 6 shows the miss rates when using different amounts of the cache as the prefetch buffer, from 64/128 to 1/128. For relatively dense matrices like `mouse_gene` and `nd24k`, very small prefetch sizes are desired. This is because their data-to-metadata ratios are high, meaning only a small amount of metadata are required to effectively determine optimized data replacements. In these cases, overly large prefetch sizes actually reduce performance for two reasons. First, a large prefetch size reduces the cache

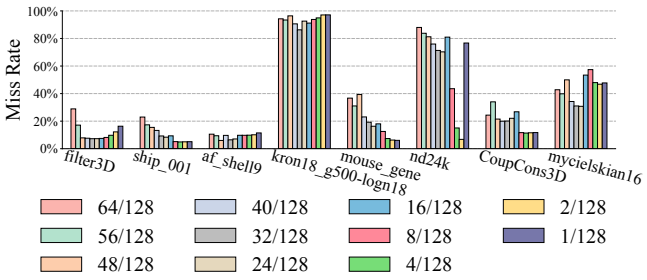


Figure 6: Miss rates when using different sizes for the guide metadata prefetch buffer as portions of the data cache.

space available for data and increases the miss rate. Second, with the gLFU implementation in Section 4.2, an excessively large prefetch size would result in many non-zero gLFU counters that exceed what the cache (including its virtual tags) can afford. Then newly incoming counters that would be useful to guide replacements cannot receive slots and be accumulated, but have to be discarded.

On the other hand, for highly sparse matrices, larger prefetch sizes are more appropriate. When the sparsity is high, more metadata are needed to effectively profile future access patterns. If the prefetch size is too small, many B fibers will not have any corresponding A elements present in the current prefetch buffer, resulting in many gLFU counters being zero and ineffective guiding.

We propose an effective mechanism to determine the prefetch size that can adapt to different matrices. We use a *two-phase* approach. Before execution starts, we use simple statistics of the sparse tensor to determine an initial prefetch size. During runtime, we further dynamically adjust the prefetch size according to the collected performance behaviors.

More specifically, we first estimate the initial prefetch size based on the sparsity of input tensors (Algorithm 1 Lines 1 to 2). Recall that the cache is mainly used for B fibers, and each B row (i.e., each A column) needs a counter. The ratio between the number of B rows and the total number of non-zeros in B (i.e., the inverse of the average non-zero length of B rows, Algorithm 1 Line 1) would be the ratio between the counter space and the B data space, which is the prefetch size as a portion of the cache capacity. To ensure the size is not too small, we empirically set a lower bound of $1/1024$, which is small enough to not affect the data caching.

Then, at runtime (Algorithm 1 Lines 3 to 18), we maintain two pairs of counters during the execution for the following two metrics. A *discard rate* is measured on the prefetch side, i.e., when incrementing counters for prefetched metadata, how many increments are discarded due to insufficient counter storage. A high discard rate indicates that too many data have been prefetched, and the prefetch size should be reduced. A *miss rate* is measured on the access side, i.e., when actually accessing the data, how many of them have no counters being maintained. A high miss rate indicates insufficient prefetching, suggesting that the prefetch size should be increased. We perform dynamic adjustments each time a certain amount of computations are performed (e.g., $1/500 \times$ total MACs), by increasing/decreasing the prefetch size when the miss/discard rate exceeds a certain threshold of 20%.

From Figure 6 we see the performance impact of the prefetch size may have multiple local optima. Thus we use simulated annealing to control the iterative adjustments. Specifically, we record the actual data access miss rates to compare across iterations. In the initial high-temperature iterations, adjustments that degrade performance may be accepted with certain probabilities. Later on, only adjustments that improve performance are accepted; otherwise, we revert to the previous prefetch size. We choose $T(k) = 0.2 \times 0.99^k$ as the temperature at iteration k, and $P(k, \Delta M) = \exp(-\frac{\Delta M}{T(k)})$ as the acceptance probability of a worse adjustment with ΔM being the increased miss rate.

The aforementioned 20% miss/discard rate threshold is empirically set and the final prefetch size is insensitive to it. We use this threshold mainly to accelerate the convergence of simulated annealing. Most cases can converge within 25 iterations and reach a size within 95% performance of the optimal selection.

Algorithm 1: Two-phase adaptive prefetch size selection.

```
// Offline phase
1  nzlB ← nnzB / (K×(J/TJ));                    // Average non-zero length of B rows
2  prefetch size ← max(1/nzlB, 1/1024);
   // Online phase
3  discard rate ← 0, miss rate ← 0;
4  for k ← 1 to 100 do
5      T(k) ← 0.2 × 0.99^k;                     // Temperature
6      Perform computations; record discard rate and miss rate;
7      if discard rate > threshold ∧ miss rate < threshold then
8          Decrease prefetch size;
9      else if discard rate < threshold ∧ miss rate > threshold then
10         Increase prefetch size;
11     else
12         Apply stochastic disturbance to prefetch size;
13     ΔM ← new miss rate − previous miss rate;
14     P(k, ΔM) ← exp(−ΔM/T(k));              // Acceptance probability
15     if P(k, ΔM) > Random() then
16         Accept new prefetch size;
17     else
18         Revert to previous prefetch size;
```

4.4 Putting It All Together

We integrate the three key techniques, namely fiber packing and splitting, practical gLFU replacement, and two-phase prefetch size selection, into SeaCache, an efficient and adaptive cache design for sparse accelerators. SeaCache only modifies the cache components, and is orthogonal to the processing element (PE) design of the accelerator; the PEs can be organized in any form, being 1D or 2D. The on-chip cache of the sparse accelerator may be potentially banked. At the high level, each bank operates independently, and its space is partitioned into two parts, for the actual data of B fibers and the prefetched guide metadata of A, respectively.

A *prefetch controller* tracks the prefetch range of A, and moves the corresponding A elements into the tail or out of the head of the prefetch buffer region in the cache, in a sliding window fashion as

described in Section 4.2. This prefetch controller also determines the prefetch size, using the two-phase mechanism proposed in Section 4.3. The prefetch size is initially set to the offline calculated size, and then periodically adjusted at runtime by temporally pausing the head/tail advances until the desired size is reached.

A *fiber iterator* is added to each PE to facilitate accesses to the packed or split fibers from the cache (Section 4.1). In the case of fiber packing, it extracts the corresponding sub-blocks from the returned full cache block. In the case of fiber splitting, it keeps the number of segments and maintains the segment index accessed so far, and calculates the adjusted fiber ID for each segment before sending the request to the cache.

Finally, *inside the cache*, we follow the structures illustrated in Figure 3 and Figure 4(b), which add extra bits and counters to each block. Furthermore, we add virtual tags in each set, and add an extra access port to the tag array, to optimize gLFU metadata updates. The tag comparators are also revised to support fuzzy matching on the lowest bits, and we may add a few more comparators to cover the additional virtual tags besides the normal ways.

Support various dataflow schemes. We have mainly discussed SeaCache with the Gust dataflow in the previous sections. On other dataflow schemes, the fiber packing and splitting techniques can be applied in the same way, as all schemes access data in the granularity of fibers. The guided replacement policies, including gLRU and gLFU, are usually used for Gust [3] which exhibits the dependent access patterns of B_k on $A_{i,k}$. The classic IP and OP schemes access the fibers in pre-determined orders and do not have these dependencies. SpArch [38], while being OP-based, further condenses the non-zeros in A to the leftmost. Therefore, when executing the outer product between A and B, the non-zeros in the condensed A column have different k coordinates and need to access different B rows in an irregular and dependent manner. As a result, the gLFU policy and the adaptive prefetch size selection can also be applied to SpArch. We evaluate this case in Section 6.3.

Support dynamic tiling. So far, we assume the input tensors are tiled offline, so the runtime hardware sees fixed tile sizes. It is also possible to use dynamic tiling [16, 17, 21] with SeaCache. Essentially, all the techniques in SeaCache work within the process of multiplying two tiles of A and B. Existing dynamic tiling designs only adjust the tile size between tiles, so the processing within a tile remains unaffected. Therefore, when integrated together, the dynamic tiling logic decides the next tile size, and then asks the PEs to fetch the corresponding fibers, which generates the accesses to the cache that will be served with the SeaCache techniques.

5 Methodology

We compare SeaCache with prior cache designs in sparse accelerators, particularly InnerSP [3], SpArch [38], and X-Cache [30]. P-OPT [4] is a CPU-based design, while its accelerator variant is just InnerSP. Table 1 summarizes the key techniques of these designs. For mapping strategies, all the three baselines use the basic fiber-ID-based mapping described in Section 2.3.1. For replacement policies, X-Cache uses standard LRU. InnerSP uses offline gLRU that pre-generates the transpose of A, whose cost is ignored. At runtime, the gLRU reuse distance is fetched and kept in addition to the cache tag in each fiber's block, and *does not* occupy the data space. In contrast, SpArch uses online gLRU that dynamically calculates the reuse distance. It uses 1/6 of the cache data space to store the head/tail pointers as well as the linked lists (Figure 1).

All the designs use the same hardware configuration: 32 MAC PEs operating at 1 GHz, with a 2 MB global cache organized into 32 banks. We also use the same dataflow for all the designs for fair comparison, though they were originally proposed under different dataflow schemes. We use Gust by default, and also evaluate condensed OP used by SpArch in Section 6.3. The cache is 16-way set-associative, with a 64-byte block size. The off-chip memory employs four DDR4 channels, providing an aggregate bandwidth of 68 GB/s. These settings are consistent with the configurations adopted in prior works [18, 31, 37]. We also evaluate performance under varying cache sizes and block sizes in Section 6.4.

To apply tiling, we mainly follow Tailors [35] and statically choose the appropriate tile size that achieves the best performance for each input matrix, under the given on-chip SRAM budget. Our tile size choices incorporate overbooking as in Tailors, and the on-demand fetching nature of caches naturally handles overbooking.

Besides caching, we also include a scratchpad baseline to demonstrate the benefits of caches over scratchpads, as described in Section 2.3. The scratchpad baseline follows the design of Tailors [35], and has the same number of PEs, on-chip SRAM capacity, and off-chip bandwidth as the above configuration.

Simulation and implementation. To evaluate performance across the different designs, we implement a cycle-accurate simulator in C++. The simulator tracks the accesses to individual non-zeros and precisely captures the impact of input sparsity patterns. Specifically, key components such as the index selector for Gust are explicitly modeled. The actual input sparse matrix is fed to the simulator to determine which data elements are actually processed in the PEs, as well as the specific fiber IDs and lengths that need to be accessed from the memory. The simulator tracks the exact tag and data contents of the cache following the proposed designs, to determine whether each access is a hit or a miss. The cache is accessed in the granularity of cache blocks. When a fiber access contains multiple cache blocks as described in Section 4.1, these blocks are accessed in a pipelined way from the cache in multiple cycles, and the access time is determined by the cache port bandwidth. In addition, we assume the access requests are issued by a decoupled controller [24] which then sends the data to all the requesting PEs. This avoids redundant accesses from multiple PEs. Our simulator is open-sourced at https://github.com/tsinghua-ideal/SeaCache-sim.

In addition, we implement the RTL designs of the key components introduced in SeaCache, including the prefetch controller, the fiber iterators, and the modified fuzzy tag comparators (Section 4.4). We synthesize these components using the Synopsys Design Compiler targeting the TSMC 28 nm technology node. The SRAM-based cache banks are modeled using CACTI 7.0 [5] to estimate the area and power. We also extend CACTI to capture the SeaCache internal modifications to the cache, including the extra bits and counters, the added virtual tags per set, and the extra read-write port in the tag array (Section 4.4). Table 2 shows the area breakdown. Most of the overheads come from the modifications to the internal cache arrays, as we add extra bits and counters per block. Nevertheless, the overall area increase is modest, less than 6%.

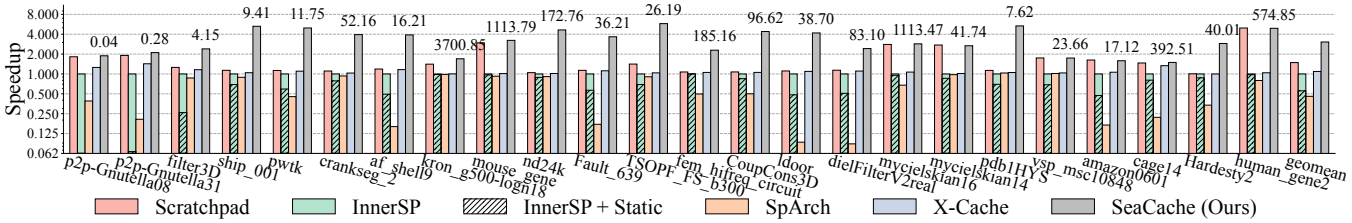


Figure 7: Performance comparison between the baselines and SeaCache, with Gust dataflow. Normalized to InnerSP. The numbers on top of the SeaCache bars represent their absolute execution time in milliseconds. The hatched regions in InnerSP indicate the performance when including its offline transpose cost.

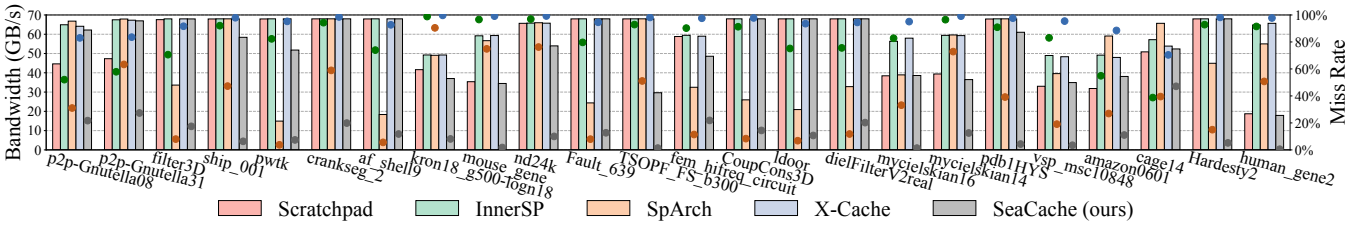


Figure 8: Memory bandwidth (bars) and miss rates (dots) of the baselines and SeaCache.

Table 2: Area breakdown of SeaCache.

Component	Area (mm^2)	Area %
32 PEs	2.80	36.78%
Interconnects	0.37	4.86%
2 MB vanilla cache	3.99	52.41%
Internal cache modifications	0.35	4.60%
All modified comparators	0.013	0.17%
Prefetch controller	0.005	0.07%
32 fiber iterators	0.085	1.12%
Total	7.613	100.00%

Datasets. We use real-world sparse matrices from the SuiteSparse Matrix Collection [9] as our evaluation datasets. These matrices cover a wide range of densities (from 0.0006% to 0.356%), non-zero counts (from 20K to 27M), and sparsity patterns. Following previous work, all matrices use 64-bit non-zero values, with 32-bit coordinates and pointers. Our primary evaluation task is sparse matrix-sparse matrix multiplication (SpMSpM) via self-multiplication of square matrices, i.e., $S \times S$, in line with prior studies. In addition, we also test several other sparse kernels, including (1) $F^T \times F$ with a tall-skinny sparse matrix F; (2) $F \times D$ where D is a random dense matrix, i.e., SpMM; (3) $F^T \times S$ as one iteration of multi-source breadth-first search (MS-BFS) in graph analytics [1, 7], where S is the graph and F represents the initial source nodes [21].

6 Evaluation

In this section, Section 6.1 first compares the overall performance among all the designs with the default configuration, and then Section 6.2 analyzes the benefits from each individual techniques of SeaCache. Section 6.3 further extends SeaCache to other accelerator dataflow schemes and sparse kernels. Finally, Section 6.4 measures the impact of key design parameters in SeaCache.

6.1 Overall Comparison

Performance. Figure 7 compares SeaCache against the scratchpad approach and the three cache baselines under the Gust PE dataflow. First, the scratchpad condenses all fibers and eliminates space fragmentation, thus achieving good overall performance with 1.48× over InnerSP. However, it is hard to apply replacement policies to scratchpads for sparse accelerators. After we evict a fiber with an arbitrary length and reclaim its space, it is difficult to re-allocate this space to another fiber with a different length. Hence, when the data exceed the buffer size, cold fibers fetched earlier will prevent hot fibers fetched later from entering the buffer. Our optimized SeaCache design can achieve similarly high space utilization while enabling gLFU replacements, thus outperforming the scratchpad.

Among the three cache baselines, each design has some advantages in some aspects but performs poorly in others. InnerSP uses a moderate cache block size and the gLRU policy, resulting in good average performance across different matrices. However, its fixed block size does not always match the matrix fiber lengths. Also, the results are optimistic. If we account for the offline transpose cost, the total execution time will increase by 1.81× on average (shown as the hatched regions), even though we use the optimized parallel transposition algorithm ScanTrans [33] with 24 threads on an Intel Xeon Gold 5120 processor at 2.2 GHz.

For SpArch with a large block size and the online gLRU policy, about half of the matrices perform well, while the rest half exhibit significantly lower performance. There are mainly three inefficiencies. First, in highly sparse matrices, such as `p2p-Gnutella31`, there are only a few elements (e.g., two) in each fiber, leaving the 576-byte block severely underutilized. Second, for those matrices preferring large tile sizes, such as `af_shell9`, the guide metadata needed

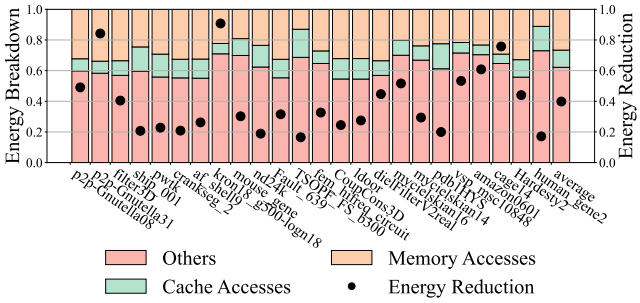


Figure 9: Energy consumption breakdown (bars) of SeaCache and its energy reduction (dots) over InnerSP.

by gLRU, including the head/tail pointers and linked list pointers, would take substantial cache space. Third, for very large matrices, such as `ldoor` and `dielFilterV2real`, the guide metadata size may even exceed the entire cache capacity. We have to reduce their tile sizes to limit these metadata to use no more than half of the cache. The smaller tile sizes will lead to worse data reuse and more redundant memory accesses [35], hence degrading performance. In summary, while gLRU theoretically gives the optimal replacement decisions, its online implementation cost is too high, requiring significant metadata storage.

X-Cache uses very short blocks and the conventional LRU policy. It does not suffer from any offline or online cost from the replacement policy, but LRU is not as good as gLRU, so its performance is lower than InnerSP. We were expecting its small block size would make it perform much worse than InnerSP, but this effect is not significant. We find that for very long fibers, even the 64-byte blocks in InnerSP still cannot efficiently cache them, leaving many data accesses to the off-chip memory.

Finally, SeaCache achieves significant speedups over all baselines, on average 2.1× over the scratchpad, 3.1× over InnerSP, 6.8× over SpArch, and 2.8× over X-Cache. Such significant improvements are enabled by the efficient fiber packing and splitting technique that better utilizes the cache block space, the gLFU policy that has cheap and practical implementation, and the two-phase adaptive prefetch size selection that prevents guide metadata from hijacking too much cache space from actual data. Essentially, SeaCache effectively addresses the inefficiencies mentioned above in the three baselines. We will later conduct in-depth performance analysis of each individual technique in Section 6.2.

Memory bandwidth and cache miss rate. Figure 8 further explains the performance gains of SeaCache using the memory bandwidth and miss rate of each design. InnerSP and X-Cache can achieve relatively low miss rates in extremely sparse matrices like `p2p-Gnutella08`, `p2p-Gnutella31`, and `amazon0601`, but have bad miss rates in other matrices. This is because the fiber lengths often exceed the block size in these not-so-sparse matrices. This leads to many cache misses, and also increases the memory bandwidth usage in these two designs. SpArch, with very large blocks, relieves this problem and reduces the miss rates for certain matrices with long fibers. But its overall miss rate and memory bandwidth remain high. We do not show the miss rate of the scratchpad since it is defined for caches, but we see that the memory bandwidth of the scratchpad is also very high and makes it memory-bound. Finally in SeaCache, both the improved mapping strategy and the replacement policy help reduce cache misses, making its miss rates the lowest among all designs. The memory bandwidth used by SeaCache also drops, sometimes even no longer memory-bound. But several highly sparse or hard-to-reuse matrices, e.g., `p2p-Gnutella08`, `fem_hifreq_circuit`, and `mycielskian16`, still exhibit high bandwidth, mainly because of the decreased execution time.

Impact of sparsity characteristics. The caching performance of sparse accelerators heavily depends on the data characteristics, among which the most important one is the *degree of sparsity*. Sparser matrices usually have shorter fiber lengths, i.e., fewer nonzeros per row. First, from the mapping perspective, matrices with short fibers suffer from cache block underutilization, and thus prefer short block sizes, e.g., `p2p-Gnutella08` for X-Cache. Matrices with long fibers suffer from cache block overflow, and need large blocks, e.g., `nd24k` for SpArch. Most matrices have diverse fiber lengths and benefit from our fiber packing and splitting design. Second, for the replacement policy, gLRU has high metadata cost (head/tail pointers and linked-list pointers) for large and sparse matrices, e.g., `af_shell9` and `ldoor` for SpArch. Third, for the prefetch size, denser matrices require smaller prefetch sizes, e.g., `mouse_gene` and `nd24k`. This is because their data-to-metadata ratios are high, meaning only a small amount of metadata would be sufficient to effectively determine the optimized data replacements.

Furthermore, the specific *sparse pattern*, i.e., the non-zero distribution, also affects the caching performance. Matrices with higher data correlations, such as power-law matrices `mycielskian16` and `amazon0601`, can still have good hit rates with a simple LRU policy as in X-Cache. Matrices with low data correlations, such as diagonal-like `ship_001` and `fem_hifreq_circuit`, perform bad under simple policies and benefit more from the guided policies, but may require large prefetch sizes.

Energy. Figure 9 shows the energy consumption breakdown of SeaCache. We mainly highlight the dynamic energy portions of cache and memory accesses. "Others" include the PE computation energy (at 3.32 W) and the static energy (at 1.02 W). Based on CACTI, we assume 0.291 nJ and 0.306 nJ for each 64-byte cache block read and write. The memory access consumes 20 pJ/bit. We see that off-chip memory accesses account for a moderate portion of energy consumption around 30%, thanks to the reduced cache misses. Overall, SeaCache reduces the energy by 60.2% compared to the InnerSP baseline.

6.2 Analysis of Individual Techniques

To separate the contribution of each technique in SeaCache, we start from a "Base" design, which uses conventional LRU and 64-byte cache blocks with basic fiber-ID-based mapping. The effects of incrementally enabling each optimization are depicted in Figure 10. By using fiber packing and splitting ("+P&S"), the space utilization of each cache block is improved for both short and long fibers, fitting more data in the cache. This achieves a 1.80× speedup.

Next, we apply the gLFU policy ("+gLFU"), using a fixed prefetch size of 1/6 cache space, same as SpArch. No virtual tags are added.

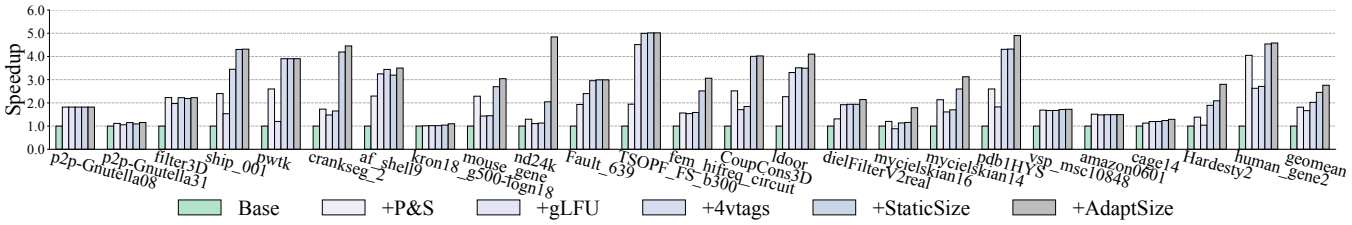


Figure 10: Performance improvements from each individual technique in SeaCache.

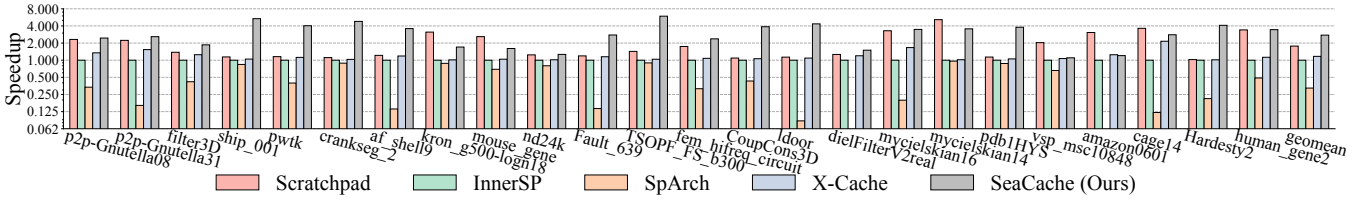


Figure 11: Performance comparison between the baselines and SeaCache, with condensed OP dataflow. Normalized to InnerSP.

However, this does not show any speedup, but degrades the performance by 7.5% on average. Only for several matrices like `af_shell9` and `dielFilterV2real`, this simple gLFU performs close to our final design. These matrices have the following characteristics. On one hand, they exhibit relatively good temporal locality, in which the prefetched fiber IDs are mostly already in the cache. So extra virtual tags are not needed. On the other hand, they fit well with the 1/6 prefetch size. Some other matrices like `CoupCons3D` also have good temporal locality, but require different prefetch sizes.

We then introduce 4 virtual tags in each 16-way set ("+4vtags"). This setting gains an overall speedup of 10.3% relative to "+P&S", but many matrices still do not have their most appropriate prefetch sizes. If we statically scan all prefetch sizes, we find that 1/16 of the cache size ("+StaticSize") has the best average speedup of 35.4% over "+P&S". Finally, we use our two-phase adaptive mechanism to select per-matrix best prefetch sizes ("+AdaptSize"). This design achieves the best performance, with 1.12× faster than "+StaticSize", and 2.75× and 1.38× over "Base" and "+4vtags".

6.3 Results of Other Dataflows and Kernels

Other dataflow. Besides Gust, SeaCache can also be used for the condensed OP dataflow [38] (Section 4.4). We compare SeaCache against the baseline scratchpad and cache designs in Figure 11 when all of them use the condensed OP dataflow. The overall performance trend is similar to that of Gust. The scratchpad only performs well on matrices that are relatively dense or have good reuse. SpArch underutilizes the large block size, especially in very sparse matrices. Several power-law matrices, including `mouse_gene` and `amazon0601`, have better reuse under condensed OP, so the relatively simple replacement policies in the scratchpad and X-Cache achieve better performance than those under Gust. Overall, with the condensed OP dataflow, SeaCache achieves average speedups of 1.6×, 2.8×, 8.6×, and 2.4× over the scratchpad, InnerSP, SpArch, and X-Cache, respectively.

Other kernels. Figure 12 evaluates other sparse kernels. In general, SeaCache still exhibits robust speedups over the four baselines,

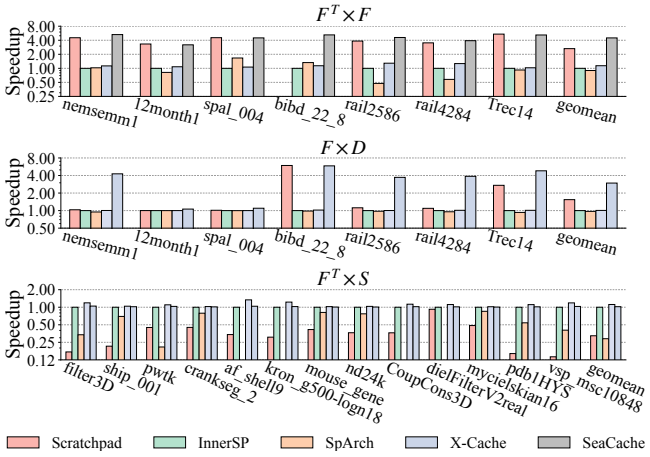


Figure 12: Performance comparison between the baselines and SeaCache, for different sparse kernels $F^T \times F$, $F \times D$, and $F^T \times S$ with Gust dataflow. The x axis shows matrix F in the first two kernels, and matrix S in the last kernel.

1.7×, 4.3×, 4.8×, 3.8× for $F^T \times F$; 1.9×, 2.9×, 3.1×, 2.9× for $F \times D$; and 3.2×, 1.0×, 3.6×, 0.9× for $F^T \times S$.

The scratchpad shows good performance in $F^T \times F$. In this kernel, the result matrix C is very small, so tiling along the k dimension and buffering the C matrix are very effective and can make data fully fit in the scratchpad. But in the cache baselines, the space utilization degrades due to underutilizing or overflowing the fixed block size, making performance suffer. Similarly, $F \times D$ on `bibd_22_8` also has a very small dense matrix D that fully fits in the scratchpad.

Note that the gains of SeaCache are also significant on the sparse-dense multiplication kernel $F \times D$. First, although D is dense, the fiber lengths of D still vary across matrices, and do not perfectly match the fixed block sizes in the baseline caches. Second, the access pattern of D still depends on the sparse F, and thus benefits from

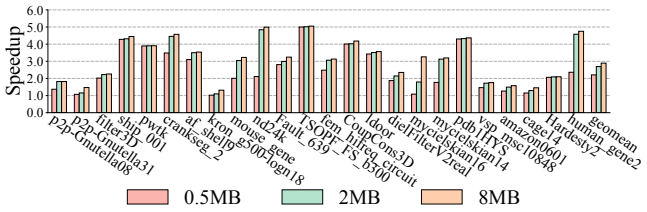


Figure 13: Impact of cache capacity on the speedups of SeaCache over the Base design in Figure 10.

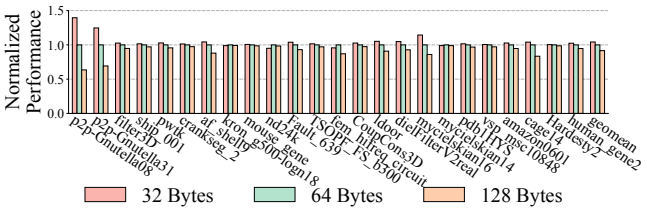


Figure 14: Impact of cache block size on performance, normalized to the default 64 bytes.

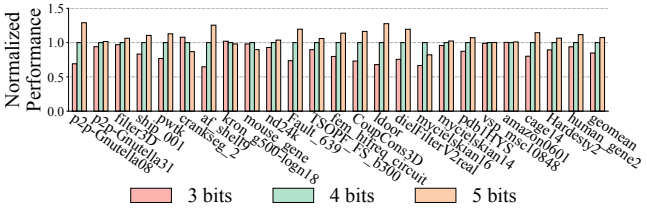


Figure 15: Impact of `Tag-L` bitwidth on performance, normalized to the default 4 bits.

the guided replacement policies. Third, the dense D matrix further reduces the overheads of prefetch metadata (e.g., one counter for a long dense row), making the guided replacement more efficient.

In the MS-BFS kernel $F^T \times S$, SeaCache's advantages are smaller. This is due to the extremely sparse matrix F^T representing the initial source nodes (one element per row), which makes the graph matrix S hardly reused, and thus the caching optimizations have limited impact. X-Cache achieves the best performance here since it uses a very small cache block size and has the fewest redundant fetches. SeaCache performs close to X-Cache.

6.4 Sensitivity Studies

Cache capacity. We measure the speedup of the full SeaCache design over the "Base" design in Section 6.2, i.e., the effectiveness of SeaCache, under different cache capacities of 0.5 MB, 2 MB (default), and 8 MB in Figure 13. We keep using 32 PEs and 64-byte blocks. SeaCache achieves consistent speedups of 2.21×, 2.69× and 2.89×. The SeaCache techniques offer more performance gains in larger capacities. The increased cache space necessitates efficient mapping strategies like fiber packing and splitting to fully utilize. Also, more space allows the gLFU policy to keep more guide metadata and thus make more accurate decisions.

Block size. Figure 14 shows the performance of SeaCache with different block sizes. We keep the 32 PEs and the total 2 MB capacity unchanged. However, when changing the block size, the access bandwidth to a bank, which is one block per cycle, also changes. We thus change the number of banks to ensure the same total bandwidth, i.e., 64 banks for 32-byte blocks, 32 banks for 64-byte blocks, etc. In such scenarios, we see the performance is not very sensitive to the block size, with smaller blocks achieving slightly better performance, i.e., 32-byte is 4.2% faster than 64-byte. However, smaller blocks need more banks, and the relative space cost of tag-to-data also increases, both adding area overheads. Thus, we choose 64-byte as our default configuration.

Number of `Tag-L` bits. Recall that in Section 4.1 we skip the lowest few `Tag-L` bits when extracting the set ID from a fiber ID. Figure 15 shows the impact of these bits. More `Tag-L` bits make more contiguous fibers mapped to the same set and allow more flexible packing, improving performance. But some matrices including `crankseg_2`, `kron_g500-logn18`, and `mycielskian16` perform worse with a longer `Tag-L`. There are two reasons. First, `Tag-L` bits are harmful to some patterned sparse matrices with high correlation between adjacent rows. If all of these rows are needed but mapped to the same set, conflicts will occur more frequently. With 16 ways per set, we can fit 16 contiguous fibers (without packing), corresponding to 4-bit `Tag-L`. Second, recall that we have a limit on the number of segments allowed when splitting fibers. A longer `Tag-L` leads to a more strict limit. Each additional `Tag-L` bit reduces the limit by half. As a result, we choose 4-bit `Tag-L`.

7 Conclusions

We propose SeaCache, a set of efficient and adaptive techniques to optimize the cache design in sparse accelerators, particularly on data mapping schemes and replacement policies. SeaCache applies fiber packing and splitting to improve the cache block space utilization in the presence of variable-length sparse fiber data. It incorporates a practical guided LFU replacement policy that has much smaller implementation cost than the optimal guided LRU but performs similarly, and also uses a two-phase adaptive mechanism to decide the proper on-chip capacity to keep the replacement policy metadata by borrowing space from the data cache. SeaCache significantly outperforms state-of-the-art cache and scratchpad designs, by 2.8× and 2.1×, respectively.

Acknowledgments

The authors thank the anonymous reviewers for their valuable suggestions, and the Tsinghua IDEAL group members for constructive discussion. Mingyu Gao is the corresponding author.

References

[1] Hasan Metin Aktulga, Aydin Buluç, Samuel Williams, and Chao Yang. 2014. Optimizing Sparse Matrix-Multiple Vectors Multiplication for Nuclear Configuration Interaction Calculations. In *28th International Parallel and Distributed Processing Symposium (IPDPS)*. 1213–1222.

[2] Alaa R. Alameldeen and David A. Wood. 2004. Adaptive Cache Compression for High-Performance Processors. In *31st Annual International Symposium on Computer Architecture (ISCA)*. 212–223.

[3] Daehyeon Baek, Soojin Hwang, Taekyung Heo, Daehoon Kim, and Jaehyuk Huh. 2021. InnerSP: A Memory Efficient Sparse Matrix Multiplication Accelerator with Locality-Aware Inner Product Processing. In *30th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. 116–128.

[4] Vignesh Balaji, Neal Crago, Aamer Jaleel, and Brandon Lucia. 2021. P-OPT: Practical Optimal Cache Replacement for Graph Analytics. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. 668–681.

[5] Rajeev Balasubramonian, Andrew B Kahng, Naveen Muralimanohar, Ali Shafiee, and Vaishnav Srinivas. 2017. CACTI 7: New Tools for Interconnect Exploration in Innovative Off-Chip Memories. *ACM Transactions on Architecture and Code Optimization (TACO)* 14, 2 (2017), 1–25.

[6] L. A. Belady. 1966. A Study of Replacement Algorithms for a Virtual-Storage Computer. *IBM Systems Journal* 5, 2 (1966), 78–101.

[7] Aydin Buluç and John R Gilbert. 2012. Parallel Sparse Matrix-Matrix Multiplication and Indexing: Implementation and Experiments. *SIAM Journal on Scientific Computing* 34, 4 (2012), C170–C191.

[8] Yu-Hsin Chen, Tien-Ju Yang, Joel Emer, and Vivienne Sze. 2019. Eyeriss v2: A Flexible Accelerator for Emerging Deep Neural Networks on Mobile Devices. *IEEE Journal on Emerging and Selected Topics in Circuits and Systems* 9, 2 (2019), 292–308.

[9] Timothy A. Davis and Yifan Hu. 2011. The University of Florida Sparse Matrix Collection. *ACM Transactions on Mathematical Software* 38, 1, Article 1 (2011), 25 pages.

[10] A. Einstein. 1916. Die Grundlage der Allgemeinen Relativitätstheorie. *Annalen der Physik* 354, 7 (1916), 769–822.

[11] Siying Feng, Xin He, Kuan-Yu Chen, Liu Ke, Xuan Zhang, David Blaauw, Trevor Mudge, and Ronald Dreslinski. 2022. MeNDA: A Near-Memory Multi-Way Merge Solution for Sparse Transposition and Dataflows. In *49th Annual International Symposium on Computer Architecture (ISCA)*. 245–258.

[12] Ashish Gondimalla, Noah Chesnut, Mithuna Thottethodi, and TN Vijaykumar. 2019. SparTen: A Sparse Tensor Accelerator for Convolutional Neural Networks. In *52nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 151–165.

[13] Kartik Hegde, Hadi Asghari-Moghaddam, Michael Pellauer, Neal Crago, Aamer Jaleel, Edgar Solomonik, Joel Emer, and Christopher W Fletcher. 2019. ExTensor: An Accelerator for Sparse Tensor Algebra. In *52nd International Symposium on Microarchitecture (MICRO)*. 319–333.

[14] Reza Hojabr, Ali Sedaghati, Amirali Sharifian, Ahmad Khonsari, and Arrvindh Shriraman. 2021. Spaghetti: Streaming Accelerators for Highly Sparse GEMM on FPGAs. In *2021 International Symposium on High-Performance Computer Architecture (HPCA)*. 84–96.

[15] Ranggi Hwang, Minhoo Kang, Jiwon Lee, Dongyun Kam, Youngjoo Lee, and Minsoo Rhu. 2023. GROW: A Row-Stationary Sparse-Dense GEMM Accelerator for Memory-Efficient Graph Convolutional Neural Networks. In *2023 International Symposium on High-Performance Computer Architecture (HPCA)*. 42–55.

[16] Jinkwon Kim, Myeongjae Jang, Haejin Nam, and Soontae Kim. 2023. HARP: Hardware-Based Pseudo-Tiling for Sparse Matrix Multiplication Accelerator. In *56th International Symposium on Microarchitecture (MICRO)*. 1148–1162.

[17] Xintong Li, Zhiyao Li, and Mingyu Gao. 2025. HYTE: Flexible Tiling for Sparse Accelerators via Hybrid Static-Dynamic Approaches. In *52nd Annual International Symposium on Computer Architecture (ISCA)*. 1613–1626.

[18] Zhiyao Li, Jiaxiang Li, Taijie Chen, Dimin Niu, Hongzhong Zheng, Yuan Xie, and Mingyu Gao. 2023. Spada: Accelerating Sparse Matrix Multiplication with Adaptive Dataflow. In *28th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 747–761.

[19] Xiaoyang Lu, Boyu Long, Xiaoming Chen, Yinhe Han, and Xian-He Sun. 2024. ACES: Accelerating Sparse Matrix Multiplication with Adaptive Execution Flow and Concurrency-Aware Cache Optimizations. In *29th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 71–85.

[20] Francisco Muñoz Martínez, Raveesh Garg, Michael Pellauer, José L. Abellán, Manuel E. Acacio, and Tushar Krishna. 2023. Flexagon: A Multi-Dataflow Sparse-Sparse Matrix Multiplication Accelerator for Efficient DNN Processing. In *28th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 252–265.

[21] Toluwanimi O Odemuyiwa, Hadi Asghari-Moghaddam, Michael Pellauer, Kartik Hegde, Po-An Tsai, Neal C Crago, Aamer Jaleel, John D Owens, Edgar Solomonik, Joel S Emer, and Christopher W Fletcher. 2023. Accelerating Sparse Data Orchestration via Dynamic Reflexive Tiling. In *28th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 18–32.

[22] Subhankar Pal, Jonathan Beaumont, Dong-Hyeon Park, Aporva Amarnath, Siying Feng, Chaitali Chakrabarti, Hun-Seok Kim, David Blaauw, Trevor Mudge, and Ronald Dreslinski. 2018. OuterSPACE: An Outer Product Based Sparse Matrix Multiplication Accelerator. In *2018 International Symposium on High Performance Computer Architecture (HPCA)*. 724–736.

[23] Angshuman Parashar, Minsoo Rhu, Anurag Mukkara, Antonio Puglielli, Rangharajan Venkatesan, Brucek Khailany, Joel Emer, Stephen W. Keckler, and William J. Dally. 2017. SCNN: An Accelerator for Compressed-Sparse Convolutional Neural Networks. In *44th Annual International Symposium on Computer Architecture (ISCA)*. 27–40.

[24] Michael Pellauer, Yakun Sophia Shao, Jason Clemons, Neal Crago, Kartik Hegde, Rangharajan Venkatesan, Stephen W Keckler, Christopher W Fletcher, and Joel Emer. 2019. Buffets: An Efficient and Composable Storage Idiom for Explicit Decoupled Data Orchestration. In *24th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 137–151.

[25] Eric Qin, Ananda Samajdar, Hyoukjun Kwon, Vineet Nadella, Sudarshan Srinivasan, Dipankar Das, Bharat Kaul, and Tushar Krishna. 2020. SIGMA: A Sparse and Irregular GEMM Accelerator with Flexible Interconnects for DNN Training. In *2020 International Symposium on High Performance Computer Architecture (HPCA)*. 58–70.

[26] Gengyu Rao, Jingji Chen, Jason Yik, and Xuehai Qian. 2022. SparseCore: Stream ISA and Processor Specialization for Sparse Computation. In *27th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 186–199.

[27] Somayeh Sardashti, André Seznec, and David A Wood. 2014. Skewed Compressed Caches. In *47th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 331–342.

[28] Somayeh Sardashti, Andre Seznec, and David A Wood. 2016. Yet Another Compressed Cache: A Low-Cost Yet Effective Compressed Cache. *ACM Transactions on Architecture and Code Optimization (TACO)* 13, 3 (2016), 1–25.

[29] Somayeh Sardashti and David A Wood. 2013. Decoupled Compressed Cache: Exploiting Spatial Locality for Energy-Optimized Compressed Caching. In *46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 62–73.

[30] Ali Sedaghati, Milad Hakimi, Reza Hojabr, and Arrvindh Shriraman. 2022. X-Cache: A Modular Architecture for Domain-Specific Caches. In *49th Annual International Symposium on Computer Architecture (ISCA)*. 396–409.

[31] Nitish Srivastava, Hanchen Jin, Jie Liu, David Albonesi, and Zhiru Zhang. 2020. MatRaptor: A Sparse-Sparse Matrix Multiplication Accelerator Based on Row-Wise Product. In *53rd International Symposium on Microarchitecture (MICRO)*. 766–780.

[32] Vivienne Sze, Yu-Hsin Chen, Tien-Ju Yang, and Joel S Emer. 2020. *Efficient Processing of Deep Neural Networks*. Springer.

[33] Hao Wang, Weifeng Liu, Kaixi Hou, and Wu-chun Feng. 2016. Parallel Transposition of Sparse Data Structures. In *2016 International Conference on Supercomputing (SC)*. 1–13.

[34] Yang Wang, Chen Zhang, Zhiqiang Xie, Cong Guo, Yunxin Liu, and Jingwen Leng. 2021. Dual-Side Sparse Tensor Core. In *48th Annual International Symposium on Computer Architecture (ISCA)*. 1083–1095.

[35] Zi Yu Xue, Yannan Nellie Wu, Joel S Emer, and Vivienne Sze. 2023. Tailors: Accelerating Sparse Tensor Algebra by Overbooking Buffer Capacity. In *56th International Symposium on Microarchitecture (MICRO)*. 1347–1363.

[36] Yifan Yang, Joel S Emer, and Daniel Sanchez. 2021. SpZip: Architectural Support for Effective Data Compression in Irregular Applications. In *48th Annual International Symposium on Computer Architecture (ISCA)*. 1069–1082.

[37] Guowei Zhang, Nithya Attaluri, Joel S Emer, and Daniel Sanchez. 2021. GAMMA: Leveraging Gustavson's Algorithm to Accelerate Sparse Matrix Multiplication. In *26th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 687–701.

[38] Zhekai Zhang, Hanrui Wang, Song Han, and William J Dally. 2020. SpArch: Efficient Architecture for Sparse Matrix Multiplication. In *2020 International Symposium on High Performance Computer Architecture (HPCA)*. 261–274.