# ACES: Accelerating Sparse Matrix Multiplication with Adaptive Execution Flow and Concurrency-Aware Cache Optimizations

Xiaoyang Lu*
xlu40@hawk.iit.edu
Department of Computer Science,
Illinois Institute of Technology
USA

Boyu Long*
longboyu21b@ict.ac.cn
Institute of Computing Technology,
Chinese Academy of Sciences
University of Chinese Academy of Sciences
China

Xiaoming Chen†
chenxiaoming@ict.ac.cn
Institute of Computing Technology,
Chinese Academy of Sciences
China

Yinhe Han†
yinhes@ict.ac.cn
Institute of Computing Technology,
Chinese Academy of Sciences
China

Xian-He Sun†
sun@iit.edu
Department of Computer Science,
Illinois Institute of Technology
USA

## Abstract

Sparse matrix-matrix multiplication (SpMM) is a critical computational kernel in numerous scientific and machine learning applications. SpMM involves massive irregular memory accesses and poses great challenges to conventional cache-based computer architectures. Recently dedicated SpMM accelerators have been proposed to enhance SpMM performance. However, current SpMM accelerators still face challenges in adapting to varied sparse patterns, fully exploiting inherent parallelism, and optimizing cache performance. To address these issues, we introduce ACES, a novel SpMM accelerator in this study. First, ACES features an adaptive execution flow that dynamically adjusts to diverse sparse patterns. The adaptive execution flow balances parallel computing efficiency and data reuse. Second, ACES incorporates locality-concurrency co-optimizations within the global cache. ACES utilizes a concurrency-aware cache management policy, which considers data locality and concurrency for optimal replacement decisions. Additionally, the integration of a non-blocking buffer with the global cache enhances concurrency and reduces computational stalls. Third, the hardware architecture of ACES is designed to integrate all innovations. The architecture ensures efficient support across the adaptive execution flow, advanced cache optimizations, and fine-grained parallel processing. Our performance evaluation demonstrates that ACES significantly outperforms existing solutions, providing a 2.1× speedup and marking a substantial advancement in SpMM acceleration.

*Both authors contributed equally to this research.
†Corresponding authors.

ACM ISBN 979-8-4007-0386-7/24/04.
https://doi.org/10.1145/3620666.3651381

**ACM Reference Format:**
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 ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3 (ASPLOS '24), April 27-May 1, 2024, La Jolla, CA, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3620666.3651381

## 1 Introduction

Sparse matrix-matrix multiplication (SpMM) is a computational process where two sparse matrices are multiplied. SpMM is a cornerstone in scientific simulations [7], linear algebra [44, 57], graph analytics [5, 6, 8, 24, 47, 59], and the rapidly evolving fields of deep learning [14, 19, 20, 33, 42]. Its crucial role in efficiently processing large-scale data structures and complex algorithms makes the effective acceleration of SpMM not just a computational challenge, but a key enabler in advancing researches and applications in these diverse and impactful domains.

The processing efficiency of SpMM is heavily influenced by the characteristics of the input sparse matrices. The high proportion of zero elements in these matrices leads to challenges such as low utilization of computational and memory resources. These irregular sparse patterns, coupled with unpredictable memory access patterns, present significant obstacles for conventional cache-based computing architectures, which are typically optimized for dense and regular data patterns. As a result, SpMM often becomes a performance bottleneck, particularly in an era where processing large datasets is increasingly crucial [15]. Given the critical role of SpMM in various computational domains, developing specialized accelerators to enhance SpMM performance is important.

Existing SpMM accelerators predominantly utilize fixed execution flows, such as inner-product (InP) [22, 45], outer-product (OutP) [43, 61], or row-by-row (ROW) [50, 60], each tailored to optimize either input or output data reuse. However, the efficiency of each execution flow is determined by sparse patterns, resulting in inconsistent performance across different sparse matrices. For instance, in InP, the sparse pattern critically influences the index intersection between the two input matrices, affecting input data fetching efficiency. In OutP, the sparse pattern determines the size of the resulting partial matrices, making output data reuse optimization crucial. In the case of ROW, the distribution of non-zeros in the input matrices influences the effectiveness of input data reuse, impacting overall data access efficiency. This variability in performance due to differing sparse patterns underscores the need for SpMM execution flow that can dynamically adapt to optimize performance across a range of matrix structures.

Additionally, the existing SpMM accelerators exhibit limited capabilities in exploiting the inherent parallelism in SpMM, leading to several inefficiencies. First, the inherent dependencies in the execution flow can lead to a collective dependency of hardware processing elements (PEs) on completing a batch of tasks. In other words, multiple PEs processing the same batch must wait for all tasks to be completed before proceeding, typically leading to the underutilization of PEs and prolonged pipeline latency. Second, synchronization challenges [50] frequently arise in existing accelerators, especially when multiple mergers are tasked with working on the same region of the output matrix. In such situations, the mergers must carefully sequence their work to ensure both the correctness and coherence of the merging process. Therefore, the merge process, responsible for consolidating the multiplication results, often becomes a bottleneck due to the need for synchronization. These inefficiencies impede the effective utilization of hardware resources and limit the overall performance of SpMM operations. This underscores the need for designs that better align execution flows with the architecture, enhance fine-grained parallelism, and mitigate the synchronization overhead, thereby maximizing the potential of parallel processing.

Furthermore, cache performance is crucial for the efficiency of SpMM accelerators, but is often overlooked. Conventional cache replacement policies used in SpMM accelerators [32, 60, 61] aim to reduce the number of cache misses but overlook the importance of concurrency. In SpMM operations, it is common for multipliers to request cache lines of a row or column concurrently. This concurrency means that even a single cache miss can cause delays in the entire processing chain. Moreover, the design of caches in current accelerators does not incorporate non-blocking features. As a result, a single cache miss causes delays in subsequent accesses, thereby exacerbating performance bottlenecks. These challenges underscore the pressing need for advanced cache optimizations in SpMM accelerators. Such optimizations must be tailored to efficiently handle the unique access demands of SpMM, considering concurrent accesses and ensuring non-blocking cache operations to optimize the overall performance of the accelerator.

In this paper, we introduce ACES, an innovative accelerator for SpMM, specifically designed to dynamically adapt its execution flow to accommodate varying sparsity patterns, optimize parallel execution, and implement locality-concurrency co-optimizations for on-chip cache. ACES has the following unique features.

- **Adaptive execution flow.** ACES is equipped with an adaptive execution flow that intelligently adjusts to varying sparsity patterns of input matrices. This adaptability is achieved through a spectrum of condensing degrees, implemented with minimal overhead and without altering the original encoding formats of the matrices. This ensures optimal performance across diverse matrix structures.
- **Balanced data reuse and synchronization.** ACES considers the reuse of input data and the synchronization needed for merging partial output results from each execution flow. By balancing memory access behavior with parallelism, it selects the most suitable condensing degree for various sparse patterns, enhancing efficiency.
- **Concurrency-aware cache management.** ACES employs PureFiber, a concurrency-aware cache replacement policy, to optimize cache management. This policy accounts for both reuse distance and potential concurrent accesses, aiming to minimize total stalls caused by cache accesses.
- **Non-blocking buffer integration.** ACES incorporates a non-blocking buffer to manage cache miss accesses, ensuring that cache misses do not significantly disrupt subsequent accesses, thereby improving data access concurrency.
- **Tailored hardware architecture.** The hardware architecture of ACES is specifically designed to complement its adaptive execution flow and cache optimizations. The architecture dedicates an addition processing element (APE) to each multiplication processing element (MPE), providing fine-grained parallelism and minimizing inter-PE dependencies.

We evaluate ACES against three state-of-the-art SpMM accelerators, including SIGMA [45], SpArch [61], and SPADA [32], across a variety of workloads with diverse sparse patterns.

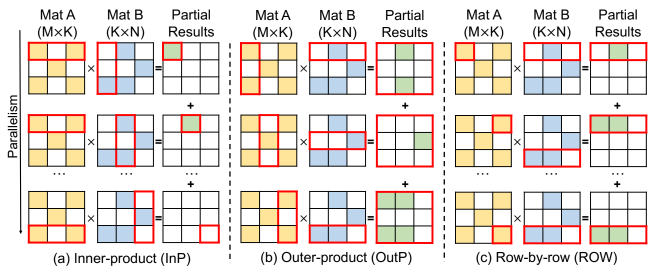

**Figure 1.** Examples of three execution flows: (a) inner-product, (b) outer-product, and (c) row-by-row.

Overall, ACES not only consistently provides optimal performance across all workloads, with average speedups of 25.5× over SIGMA, 8.9× over SpArch, and 2.1× over SPADA, but also achieves this with the lowest area cost, further underscoring its efficiency and innovation in SpMM acceleration.

## 2 Background

### 2.1 Execution Flows for SpMM

The multiplication of two sparse matrices involves specific execution flows, dictating how the matrices are accessed and processed. The three primary execution flows in SpMM are InP [22, 45], OutP [43, 61], and ROW [50, 60]. Figure 1 demonstrates how matrices **A** and **B** are multiplied to produce output matrix **C** using each execution flow. Each execution flow exhibits distinct characteristics in terms of input and output reuse, index intersection efficiency, and synchronization requirements.

InP computes each element of matrix **C** by calculating the inner product of a corresponding row of matrix **A** and a column of matrix **B**. InP achieves full reuse of matrix **C**, as all partial results for each element are immediately accumulated. Additionally, InP allows multiple PEs to compute different elements of the output matrix simultaneously, thus minimizing synchronization issues. However, it faces challenges in efficiently reusing matrix **B**, since each column of matrix **B** must be refetched multiple times for every row in matrix **A**. Moreover, InP often encounters inefficiencies in index intersection due to the sparsity of matrices **A** and **B**. When fetching an entire row of **A** and an entire column of **B** for computation, only non-zero elements with matching indices contribute to **C**. Given the sparsity, many fetched elements lack matching indices, leading to numerous redundant operations. This inefficiency is exemplified in Figure 1(a), where the index intersection between the last row of matrix **A** and the last column of matrix **B** is illustrated.

OutP computes the outer product of a row of matrix **A** with a column of matrix **B**, forming one partial matrix of **C** at a time. These partial matrices are subsequently merged to produce the final output matrix **C**. OutP allows for efficient reuse of the input matrices, as each row of **A** and each column of **B** are fetched only once. Additionally, OutP mitigates the inefficiencies in index intersection seen in InP, since only entirely zero columns of **A** and rows of **B** do not contribute to the output, a scenario that is relatively rare. However, OutP faces challenges in managing the size and number of partial output matrices, particularly when dealing with matrices that have highly irregular sparsity patterns. The cumulative size of all partial matrices often surpasses that of the final output matrix, leading to significant memory traffic and computational complexity during the merging process. This issue becomes even more pronounced when handling large matrices, where the memory and computational demands are substantially increased. Furthermore, when multiple PEs are involved in generating and merging these partial matrices, synchronization becomes necessary to ensure a correct merging process. A large number of synchronization requirements can significantly limit the efficiency of the accelerator and the utilization of PEs.

ROW operates based on row-wise partitioning of the input matrices. For each row $\mathbf{C}_i$ of the output matrix **C**, ROW calculates the result by merging the scalar-vector products of each non-zero element $a_{i,k}$ in row $\mathbf{A}_i$ with the corresponding row $\mathbf{B}_k$ of matrix **B**. ROW generates and stores partial results for a single output row at a time, enabling efficient on-chip management and good reuse of matrix **C**. In terms of index intersection between the two input matrices, ROW is efficient, as it fetches only those pairs of non-zeros that are already matched. However, ROW faces challenges in efficiently reusing the rows of matrix **B** due to irregular access patterns driven by the non-zero distribution in matrix **A**. Moreover, when PEs work concurrently to generate partial results for the same output row, synchronization becomes necessary, as depicted in Figure 1(c), potentially creating bottlenecks in highly parallel systems.

While conventional execution flows in SpMM provide distinct characteristics to matrix multiplication, there is no universally optimal solution. The complexity of handling highly irregular sparsity patterns, coupled with the demands of parallel computing, underscores the need for innovative, adaptive execution flows that can efficiently balance data reuse, index intersection, and parallelism.

### 2.2 SpMM Accelerators

SIGMA [45] is an InP-based SpMM accelerator that enhances index intersection efficiency through a bitmap format for sparse matrix representation. This format ensures that only essential computations are conducted. Additionally, SIGMA employs flexible interconnections among PEs to optimize their utilization. SpArch [61] adopts an OutP execution flow for SpMM, specifically focusing on enhancing the efficiency of the merging process. It proposes an aggressively condensed matrix representation for matrix **A**, which reduces the number of partial matrices produced during multiplication but may compromise input reuse. Additionally, SpArch integrates a high-radix merger, pipelining the production and merging of partial matrices. SPADA [32] inherits ROW and introduces a window-based adaptive (WA) execution flow to effectively adapt SpMM to various sparse patterns, thereby addressing the limitations of traditional fixed execution flows. The specialized hardware architecture of SPADA includes multiple multipliers, which are responsible for concurrently executing scalar-vector multiplications within a WA window. However, WA introduces a collective dependency among the multipliers. This dependency dictates that a multiplier can only proceed to the next window once all the multiplication tasks in the current window are completed.

### 2.3 Challenges and Opportunities

SpMM accelerators with fixed execution flows, such as SIGMA [45] and OuterSPACE [43], have introduced innovations in accelerating SpMM. However, they also encounter challenges due to the limitations of their fixed execution flows. In contrast, the WA execution flow of SPADA [32] offers an innovative approach to adapt to diverse sparse patterns. However, WA relies on collective dependencies among multipliers for coordinated execution, which, while essential, introduces potential bottlenecks in hardware parallelism. This dependency can limit the efficiency and scalability of the system, especially in cases where high parallel processing is required.

Moreover, the performance of on-chip caches presents a significant challenge in the context of concurrent data demands inherent in SpMM operations. Current accelerators [32, 60, 61] optimize cache performance on data locality. However, data concurrency is equally important [36, 38, 46, 52]. Effective cache management should consider data locality and concurrency together. In addition, the on-chip cache should handle cache misses in a non-blocking manner, allowing the cache to continue servicing other requests efficiently. The non-blocking design could enhance the data concurrency and reduce computational stalls.

Leveraging insights from the existing accelerators and the characteristics of SpMM, ACES differentiates itself with an adaptive execution flow that balances data reuse and parallelism, which effectively overcomes the limitations of fixed execution flows and the collective dependencies of WA. Furthermore, ACES emphasizes the co-optimizations of locality and concurrency in on-chip cache design and management. We introduce the details of ACES in the subsequent section.

## 3 ACES Architecture

### 3.1 Overview of ACES

Figure 2 presents an overview of ACES, an accelerator designed for efficient SpMM. The key components of ACES consist of a condensing adaptor to dynamically determine the execution flow; multiple PEs, including MPEs for handling multiplications and APEs for merging partial results; two schedulers (synchronization scheduler and merging scheduler) that adaptively distribute tasks across APEs; and a

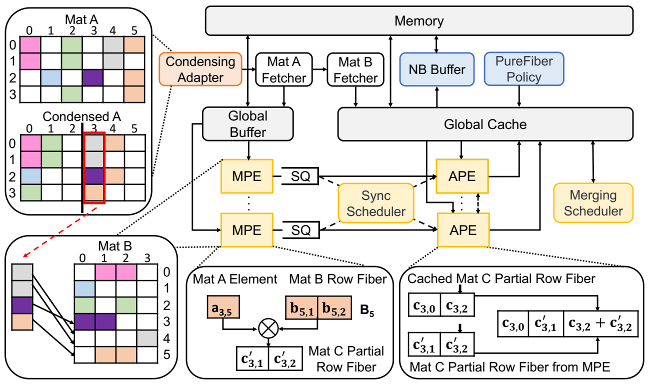

**Figure 2.** Overview of ACES.

global cache, which is integrated with a non-blocking buffer (NB buffer) to organize a non-blocking cache system. These components synergize to build the key features of ACES: an adaptive execution flow, a tailored hardware architecture, and advanced locality-concurrency co-optimizations in cache management.

**Adaptive execution flow.** ACES incorporates a condensation adapter that dynamically tunes the condensed matrix representation for matrix **A** to optimize the execution flow, as depicted in the top left of Figure 2. This feature allows for flexibility in handling varying sparsity patterns of matrix **A** (more details in Section 3.2). A fetcher for matrix **A** retrieves elements from the condensed columns, placing them in a global buffer, specifically designed as a lightweight buffer to store elements of matrix **A**, while another fetcher fetches necessary rows of matrix **B** into the global cache.

**Tailored hardware architecture.** ACES employs parallel computing, utilizing MPEs and APEs to support its adaptive execution flow (detailed in Section 3.3). Each MPE, illustrated in the bottom left of Figure 2, loads a distinct non-zero element from the condensed column of matrix **A** and a corresponding row from matrix **B**. These MPEs execute scalar-vector multiplications in parallel, as shown in the bottom middle of Figure 2. In conjunction with each MPE, each APE independently performs immediate merging of the partial output rows produced by the MPE with the corresponding partial result stored in the global cache, as demonstrated in the bottom right of Figure 2. A synchronization scheduler is designed to mitigate synchronization conflicts between APEs when processing immediate merging (more details in Section 3.4). After completing all multiplication operations, the APEs, under the coordination of a merging scheduler (detailed in Section 3.4), engage in the final merging stage to merge the remaining partial results into the final output matrix.

**Locality-concurrency co-optimizations for global cache.** The global cache in ACES is a critical component, designed to efficiently manage both matrix **B** rows and matrix **C** partial output rows. The PureFiber cache replacement policy is

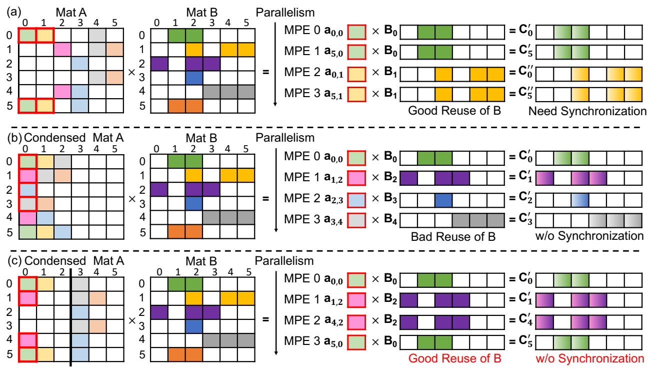

**Figure 3.** Three execution flows with different condensing degrees: (a) w/o condensing, (b) aggressive condensing, and (c) moderate condensing.

employed to manage cache lines effectively by considering both data locality and concurrency (more details in Section 3.5). The integration of a NB buffer in the global cache (more details in Section 3.6) supports the PureFiber policy and enhances performance by facilitating non-blocking accesses.

## 3.2 Adaptive Execution Flow

Inspired by ROW, in ACES, each MPE is tasked with a scalar-vector multiplication, involving a single non-zero element from matrix **A** and the corresponding row in matrix **B**. This design addresses the inefficiencies of index intersection in InP and reduces the collective dependency among MPEs. Moreover, generating partial results at row granularity offers several advantages. It supports immediate merging, which pipelines both multiplication and merging processes. Additionally, it alleviates the challenges in OutP, where transferring entire partial matrices during the merging stage can significantly increase memory traffic. However, the ROW method compromises the reuse of matrix **B**, as non-zero elements from the same row of matrix **A** are often processed concurrently by MPEs, leading to requests for different rows of matrix **B**. To optimize the reuse of matrix **B**, we traverse the non-zero elements of matrix **A** column by column, similar to OutP. Figure 3(a) demonstrates this execution flow when no condensing is applied to matrix **A**. While this execution flow enhances the reuse of matrix **B**, it can also lead to multiple MPEs generating partial results for the same output matrix row, particularly when matrix **A** has high sparsity. This situation introduces synchronization challenges among APEs during the immediate merging.

An aggressive condensed matrix representation for matrix **A** involves shifting all non-zero elements to the leftmost columns, which results in much denser condensed columns. Despite condensation, each non-zero element retains a record of its original column index to ensure computational accuracy [61]. With the aggressive condensed matrix representation, traversal of non-zero elements of matrix **A** occurs through condensed columns, impacting the execution flow of SpMM. In terms of implementation, matrix **A** is stored in the CSR format, where each row is an ordered list of column indices. The elements sharing the same index in each ordered list are fetched and assigned to MPEs for parallel computing. As shown in Figure 3(b), when four MPEs concurrently perform scalar-vector multiplications using elements from a condensed column of matrix **A** and corresponding rows from matrix **B**, the potential for reusing rows from matrix **B** diminishes due to simultaneous requests for different rows. However, aggressive condensing often generates distinct partial rows for the output matrix during parallel scalar-vector multiplications. This results in a diminished need for synchronization among APEs during immediate merging.

In an effort to strike a balance between data reuse and parallel computing efficiency, we introduce a moderately condensed matrix representation for matrix **A**. As depicted in Figure 3(c), the columns of matrix **A** are divided into two distinct groups: the first half in one group and the second half in the other. Within each group, non-zero elements of matrix **A** are shifted to the leftmost columns, which is the same as in aggressive condensing. For the example in Figure 3, the moderate condensing of matrix **A** enhances the reuse of matrix **B** when compared with aggressive condensing. Furthermore, it reduces synchronization challenges that might emerge without condensing.

Each condensing degree tries to offer a trade-off between data access and parallelism. However, given the complex and varied sparsity patterns in matrices, no single condensing degree can consistently outperform others across all workloads. This highlights the critical need for an adaptive mechanism, tailored to dynamically adjust to the unique sparse pattern of each input, thereby optimizing overall performance. ACES introduces such a mechanism, offering a spectrum of condensing degrees to ensure an optimized execution flow tailored to each workload.

In ACES, we employ a condensing adapter to dynamically adjust the representation of matrix **A** among three condensing degrees: none, moderate, and aggressive. We first partition the entire matrix into bands. Drawing on insights from [32], we recognize that adjacent rows with similar distributions of non-zero elements tend to have a stable row length (number of non-zero elements). Leveraging this observation, the partitioning of the matrix into bands is primarily based on row lengths, as indicated by the CSR offsets array, ensuring a fast and lightweight partitioning. A new band is established whenever the absolute difference in row length between adjacent rows surpasses a specified threshold, which we empirically set to be 10. Considering that each band displays a similar sparse pattern, we select the optimal condensing degree for each. For large bands, consisting of at least 256 rows, we identify the optimal condensing degree through the sampling phase. In the sampling phase, we execute three sample passes, each containing 32 rows. During these passes, we execute the SpMM with different condensing degrees and monitor the overall execution time, including both multiplication and immediate merging tasks. The condensing degree resulting in the best execution time is then applied to the remaining rows in the band, as it offers the most efficient balance between data reuse and parallelism. For small bands, we apply moderate condensing by default due to insufficient data for sampling, striking a balance between no and aggressive condensing.

Once the execution flow is determined, the fetcher for matrix **A** loads non-zero elements of matrix **A** into the global buffer ahead of execution. Concurrently, the fetcher for matrix **B** rows retrieves the row corresponding to the matrix **A** element being fetched and places it in the global cache. Each row of matrix **B**, stored as a fiber [32, 60], is a list sorted by coordinate, consisting of the coordinates and values of each non-zero element. Then, the element from matrix **A** and the corresponding fiber of matrix **B** are dispatched to an available MPE.

## 3.3 Processing Elements

Each MPE in ACES is specifically designed for executing scalar-vector multiplications, a fundamental operation in SpMM. The partial output fiber produced by an MPE is filled into a corresponding selective queue (SQ), buffers the sequentially generated products from the multiplier. Due to the independence of each scalar-vector multiplication, every MPE can operate concurrently, avoiding collective dependencies and thereby maximizing processing element utilization. ACES adopts a distinctive one-to-one pairing of MPEs with APEs, facilitating efficient processing of different rows of the output matrix in SpMM. The bottom middle of Figure 2 shows the example of MPE executes the scalar-vector multiplication between $a_{3,5}$ with the corresponding row $\mathbf{B}_5$.

In tandem with MPEs, APEs play a crucial role, particularly in the immediate merging of partial output fibers. Each APE handles the merging of newly produced partial output fibers from SQ with corresponding partial output fibers previously generated and stored in the global cache. This design allows APEs to initiate the merging process as soon as a partial fiber becomes available in the corresponding SQ, thus enhancing the efficiency of the system. The merging of each fiber is achieved by walking two pointers over the two fibers, comparing the corresponding coordinates, merging them accordingly if the coordinates match, and then advancing the pointers based on the comparison results. The outcome of this merging is a new fiber of matrix **C**, which is a sorted merge of the two input fibers and is then written back to the global cache for further processing. The bottom right of Figure 2 illustrates an APE executing the merging between two partial fibers. In instances where there is no partial fiber in the global cache that can be merged with the partial fiber loaded from the SQ, the APE will write the partial fiber into the cache directly. This approach is implemented to prevent the stalls that could arise from waiting for a matching fiber to be fetched from memory. Immediate merging at row granularity in ACES facilitates the easy storage of partial fibers in the global cache and ensures effective reuse of matrix **C**, consequently reducing memory traffic. The independent and concurrent processing by the APEs, in collaboration with the synchronization scheduler, enhances system parallelism and optimizes the utilization of MPEs. Meanwhile, the MPEs continue with subsequent multiplication tasks, further amplifying the parallel processing capabilities of ACES.

In immediate merging, APEs focus on merging newly generated partial fibers with those currently stored in the global cache. However, due to the limited capacity of the global cache, it is not feasible to keep all partial fibers there. Consequently, some partial fibers are periodically written back to DRAM. This necessitates a final merging stage after the completion of all scalar-vector multiplications. During the final merging, every APE in the system participates, with each APE merging two partial fibers at a time. ACES employs a merging scheduler to optimize this final merging process, ensuring the final output matrix is assembled efficiently.

## 3.4 Schedulers

In ACES, two schedulers are introduced: the synchronization scheduler, which strives to streamline the immediate merging process and reduce synchronization delays, and the merging scheduler, managing the final merging stage to minimize memory accesses.

**Synchronization scheduler.** Once the partial fibers are generated by MPEs, they are buffered in the SQs. ACES utilizes the synchronization scheduler to efficiently assign fibers to APEs for initiating the immediate merging process. The synchronization scheduler coordinates with SQs and APEs. It tracks the rows of partial fibers that the APEs are currently merging to schedule subsequent merge tasks that minimize stalls of the APEs due to synchronization issues.

When an APE becomes available, the synchronization scheduler evaluates the head fiber of the corresponding SQ. The design of the SQ acts similarly to a normal FIFO, but allows for selective access to stored fibers, enabling the scheduler to select an appropriate fiber to minimize synchronization conflicts. If the top fiber in an SQ cannot be processed due to another MPE currently updating the corresponding row in the cache, the scheduler selects the next available fiber that does not have a synchronization issue. This approach ensures continuous processing and minimizes idle time for the APEs. In cases where the top fibers in different SQs correspond to the same row, the scheduler randomly picks an available APE from the corresponding APEs to first merge these two partial fibers. After this initial merge, the APE then merges the result with the corresponding partial fibers stored in the cache. For scenarios without synchronization conflicts, the scheduler assigns the top fibers from

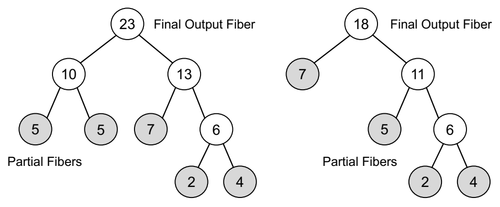

**Figure 4.** Example of Huffman trees for two different rows of the output matrix.

the SQs directly to their corresponding APEs. With the help of the synchronization scheduler, execution stalls caused by synchronization issues are mitigated, and the parallelism of APEs is guaranteed.

**Merging scheduler.** In the final merging stage, the merging scheduler aims to minimize memory accesses. Drawing inspiration from SpArch [61], ACES adapts the Huffman tree, traditionally used in data compression for minimizing the total weighted path length of encoded symbols [26], to orchestrate the merging process for each row of the output matrix. Each Huffman tree in ACES is a binary tree, with leaf nodes representing partial fibers from a specific row of the output matrix. The weight of each node equates to the number of non-zero elements in the fiber it represents. The merging of fibers with the lowest weights forms internal nodes, each representing a merged result. The root node represents the fully merged fiber for that row. The bottom-up construction of the Huffman tree in ACES prioritizes the merging of fibers with fewer elements first. This approach not only reduces the number of memory loads and stores required during the final merging phase, but also decreases the total number of comparisons and operations needed, leading to a more efficient merging process. Figure 4 provides a simplified representation that assumes no intersection of non-zero elements between the leaf nodes, where the weight of each internal node is the sum of the weights of its child nodes. In practice, however, the actual number of non-zero elements post-merging is often lower due to the presence of intersections.

In the practical implementation of ACES, the Huffman tree is constructed using a priority queue. Initially, weights of leaf nodes, each representing a partial fiber, are entered into the queue. In each iteration, the two fibers with the lowest weights are extracted from the queue and merged, forming a new task encapsulated as a vector of fibers. With the updated weight, the merged fiber is then reinserted into the priority queue for potential future merging operations. This process repeats until all partial fibers corresponding to an output row are merged into a single fiber. Upon completing the merging tasks for one row, the priority queue is efficiently repurposed for the next, reducing the need to maintain a separate queue for each row. The merging scheduler holds the determined tasks temporarily in a small buffer, preserving the order in

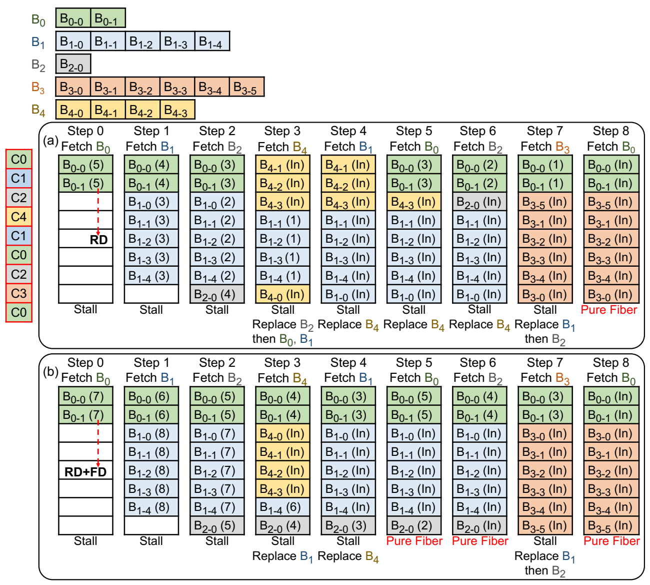
$B_0$ $B_{0\text{-}0}$ $B_{0\text{-}1}$
$B_1$ $B_{1\text{-}0}$ $B_{1\text{-}1}$ $B_{1\text{-}2}$ $B_{1\text{-}3}$ $B_{1\text{-}4}$
$B_2$ $B_{2\text{-}0}$
$B_3$ $B_{3\text{-}0}$ $B_{3\text{-}1}$ $B_{3\text{-}2}$ $B_{3\text{-}3}$ $B_{3\text{-}4}$ $B_{3\text{-}5}$
$B_4$ $B_{4\text{-}0}$ $B_{4\text{-}1}$ $B_{4\text{-}2}$ $B_{4\text{-}3}$
C0 C1 C2 C4 C1 C0 C2 C3 C0
(a)
Step 0 Fetch $B_0$
Step 1 Fetch $B_1$
Step 2 Fetch $B_2$
Step 3 Fetch $B_4$
Step 4 Fetch $B_1$
Step 5 Fetch $B_0$
Step 6 Fetch $B_2$
Step 7 Fetch $B_3$
Step 8 Fetch $B_0$
RD
Stall
Replace $B_2$ then $B_0$, $B_1$
Replace $B_4$
Replace $B_4$
Replace $B_4$
Replace $B_1$ then $B_2$
Pure Fiber
(b)
RD+FD
Replace $B_1$
Replace $B_4$
Pure Fiber
Replace $B_1$ then $B_2$

**Figure 5.** Comparison of cache replacement policies: (a) behavior of Belady's OPT policy, and (b) behavior of PureFiber.

which they were created. When an APE becomes available, the merging scheduler prioritizes selecting the next task that minimizes synchronization conflicts. The merging scheduler adheres to the optimal merging order prescribed by the Huffman trees and mitigates synchronization risks, enhancing the overall processing efficiency in the final stage of SpMM.

## 3.5 Global Cache and PureFiber Policy

The global cache in ACES, organized as a multi-banked, set-associative cache, stores fibers of matrix **B** and partial output fibers of matrix **C**, supporting cache line granularity accesses for flexible capacity sharing among various fibers. During MPE operation in ACES, multiplying an element from matrix **A** with elements from the corresponding row in matrix **B** leads to concurrent requests for cache lines of matrix **B**. A single cache miss among these requests can stall the scalar-vector multiplication process, as the MPE needs all required data from matrix **B** to produce a complete partial fiber. We define **pure fiber** as a scenario where cache lines of a fiber are accessed concurrently without any cache misses. Achieving a high number of pure fibers in SpMM is crucial for mitigating cache stalls. Contrasting with traditional accelerators [32, 61], which mainly focus on reducing cache misses, we have developed PureFiber, a concurrency-aware cache replacement policy designed to prioritize achieving more pure fibers in SpMM.

PureFiber integrates data locality and concurrency considerations for each cache line when making eviction decisions. It employs the *Next Request Distance (RD)*, dynamically capturing the reuse distance to the fiber, which indicates the expected time until the fiber is next requested. The RD value is initialized upon cache line insertion or a cache hit and is decremented until the line is reused, thereby serving as a measure of temporal locality. Additionally, PureFiber evaluates the *Fiber Density (FD)*, representing the number of cache lines in the corresponding fiber and serving as an indicator of potential concurrent accesses. When making eviction decisions, PureFiber selects the cache line with the highest combined sum of RD and FD for eviction. If multiple candidates are present, PureFiber prioritizes evicting the line with higher fiber density. By balancing data locality with concurrency, PureFiber optimizes cache management, focusing on increasing the number of pure fibers to support uninterrupted computations and enhance overall performance.

Figure 5 presents a simplified case study, all the cache capacity is used to store lines of matrix **B**. The leftmost column represents a condensed column from the condensed **A** matrix. Each element is annotated with its original column number. For instance, the element labeled $C0$ originates from column 0 in the original matrix. Figure 5 top displays row fibers of matrix **B**, segmented according to the cache line length, as indicated at the top of the figure ($\mathbf{B}_0$ to $\mathbf{B}_4$). In this example, the cache can store at most 8 lines.

Figure 5(a) illustrates Belady's OPT policy [4], which focuses solely on temporal locality. This policy prioritizes evicting cache lines with the largest RD values. From timestamps 0 to 2, lines from $\mathbf{B}_0$, $\mathbf{B}_1$, and $\mathbf{B}_2$ are loaded into the cache. At timestamp 3, to accommodate lines from $\mathbf{B}_4$, the policy first evicts lines from $\mathbf{B}_2$ and $\mathbf{B}_0$ because they have larger RD values than those from $\mathbf{B}_1$. Then, to fully accommodate $\mathbf{B}_4$, line $\mathbf{B}_{1\text{-}0}$ is also evicted. At timestamp 4, the miss of a single line from $\mathbf{B}_1$, despite four hits, leads to a stall in output fiber production. As demonstrated in Figure 5(a), Belady's policy results in only one pure fiber. Figure 5(b) illustrates the cache replacement decisions made by PureFiber under the same access pattern. In a non-blocking cache system capable of handling concurrent misses (detailed in Section 3.6), PureFiber achieves three pure fibers. By considering data locality and prioritizing the retention of fibers with lower density, such as $\mathbf{B}_0$ and $\mathbf{B}_2$, PureFiber secures two additional pure fibers at timestamps 5 and 6. This example underscores the ability of PureFiber to enhance performance by balancing considerations of locality and concurrency.

In ACES, PureFiber manages the cache lines for fibers in matrix **B** as well as the partial output fibers of matrix **C**. By prioritizing the retention of output fibers with smaller RD values, PureFiber increases the probability that corresponding partial results previously generated remain available in the cache for immediate use. Considering the locality of the partial output fibers of matrix **C**, APEs are able to participate in immediate merging, which enhances their utilization and reduces the number of tasks needed in the final merging stage. Additionally, the retention of fibers with smaller FD values helps prevent the APEs from engaging in time-consuming immediate merging tasks, especially when an MPE generates a low-density fiber that requires to be merged immediately with a much denser partial fiber already in the cache. By prioritizing the eviction of higher-density fibers, PureFiber improves the overall pipeline efficiency in ACES.

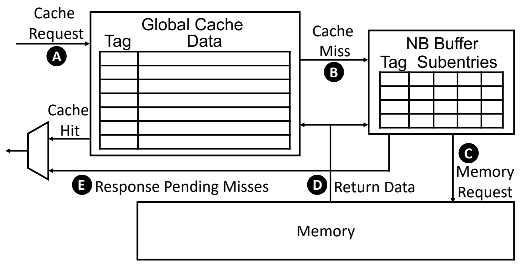

**Figure 6.** Organization of the non-blocking cache.

## 3.6 Non-Blocking Buffer

Traditional accelerators strive for speedups through extensive parallel computation. However, the effectiveness of parallelism is often constrained by the performance limitations of the memory system. In particular, a conventional blocking global cache stalls upon a cache miss, waiting until the missing cache line is retrieved from memory. This behavior significantly hinders SpMM performance due to its irregular access patterns and frequent cache misses.

In response, ACES integrates an NB buffer with the global cache, creating an efficient non-blocking cache system. The non-blocking cache [29], widely adopted in modern processors, ensures that cache misses do not stall subsequent cache accesses. A non-blocking cache can handle a number of outstanding misses, significantly improving data concurrency [2, 38, 46]. Figure 6 shows the organization and workflow of the non-blocking cache in ACES. For each global cache request (A), when a miss occurs, the corresponding miss information is stored in an entry of the NB buffer (B). NB buffer tracks all outstanding cache misses, with each entry referring to one missing cache line and containing multiple subentries for handling multiple misses to the same line. If it is the first miss to a specific cache line, a memory request is issued (C); subsequent misses for the same line are consolidated within the corresponding entry. While the cache continues to service other requests, the NB buffer fetches the missing data from the main memory. Once the data is retrieved, the buffer updates the global cache with the new data and resolves all associated misses (D). Once the misses are resolved, the corresponding entry in the NB buffer is released for future cache misses. All PEs or fetchers waiting for that cache line are then notified that the data is available in the cache (E). By allowing the cache to handle other requests during miss processing concurrently, the non-blocking design in ACES significantly reduces memory stall cycles and enhances concurrency.

**Table 1.** Configuration of ACES.

| | |
|---|---|
| MPEs | 16 MPEs (multipliers); 1 GHz |
| APEs | 16 APEs (merger); 1 GHz |
| SQs | 16 SQs, 2 KB per queue |
| Global Buffer | 0.5 KB, 32-entry buffer |
| Global Cache | 1 MB, 16 banks, 16-way associative |
| Crossbar | 16×16 and 16×16, swizzle-switch based |
| NB buffer | 0.5 KB, 64 subentries |
| Memory | 128 GB/s, 16 64-bit HBM channels, 8GB/s per channel |

**Table 2.** Evaluated workloads.

| Workload | Density | Workload | Density |
|---|---|---|---|
| 2cubes_sphere (cs) | 1.6e-04 | offshore (of) | 6.3e-05 |
| amazon0312 (az) | 2.0e-05 | p2p-Gnutella31 (pg) | 3.8e-05 |
| ca-CondMat (cc) | 3.5e-04 | patents_main (pm) | 9.7e-06 |
| cage12 (cg) | 1.2e-04 | poisson3Da (p3) | 1.9e-03 |
| cop20k_A (ca) | 1.8e-04 | roadNet-CA (rc) | 1.4e-06 |
| email-Enron (ee) | 2.7e-04 | scircuit (sc) | 3.3e-05 |
| filter3D (f3) | 2.4e-04 | web-Google (wg) | 6.1e-06 |
| m133-b3 (mb) | 2.0e-05 | webbase-1M (w1) | 3.1e-06 |
| mario002 (m2) | 1.4e-05 | wiki-Vote (wv) | 1.5e-03 |

## 4 Evaluation Methodology

Table 1 presents the detailed configuration of ACES. The area of ACES is measured by writing RTL for core components, including MPEs, APEs, the condensing adapter, and schedulers, and synthesizing them using Synopsys Design Compiler on the TSMC 28 nm technology. CACTI 7.0 [3] is used to model the overhead of global cache, global buffer, and NB buffer. For interconnections, we follow the approach used in previous work [32] and model swizzle-switch networks [48] to connect banks of the global cache with PEs, thereby facilitating concurrent accesses. Furthermore, a cycle-accurate simulator is built to accurately measure performance, PE utilization, and memory traffic. This simulator is utilized to model interactions between hardware components and implement ACES adaptive execution flow.

We evaluate the performance of ACES using the SuiteSparse matrix collection [12], which is a widely recognized benchmark in prior works [32, 61]. Table 2 presents a selection of 18 sparse matrices from this dataset, chosen for their wide range of sparse patterns and densities. The diverse set of matrices provides a comprehensive basis for evaluating the adaptability and efficiency of ACES across varying matrix characteristics. To construct SpMM workloads, we adhere to the methodology outlined in SPADA, wherein a square matrix is multiplied by itself and a non-square matrix is multiplied by its transpose, ensuring consistency and fairness in our evaluation.

For comparison, we selected three state-of-the-art SpMM accelerators: SIGMA [45], SpArch [61], and SPADA [32]. SIGMA, an InP-based accelerator, and SpArch, which adopts the OutP execution flow, are chosen to represent two fundamental execution flows in SpMM accelerators. SPADA is included for its adaptive execution flow, which incorporates the ROW execution flow as one of its modes, offering a comprehensive perspective for comparison. To ensure fair comparisons among all accelerators, we standardized the hardware configurations. The number of multipliers was aligned to 16, following the configurations of SpArch and SPADA. For SIGMA, we scale down the Flex-DPE to a width of 16, reduce the SRAM buffer size to 1.5 MB, and increase the operating frequency to 1 GHz. Furthermore, each accelerator utilizes the same HBM module for off-chip data storage. The data precision across all accelerators is standardized to 64-bit double-precision, which is a common requirement in scientific computing applications [32]. Regarding input formats, while ACES processes inputs in the CSR format, the other accelerators use their respective recommended formats, ensuring optimal operational conditions for each. When evaluating the performance of ACES, we include the cost associated with the sampling phases, ensuring a comprehensive and transparent performance comparison.

## 5 Experiment Results

### 5.1 Performance Comparisons

Figure 7 illustrates the performance of various SpMM accelerators, normalized to that of SpArch, across all evaluated workloads. We make four major observations. First, ACES consistently outperforms all state-of-the-art accelerators across every workload, highlighting its exceptional adaptability to diverse sparse patterns. On average, ACES achieves significant performance gains of 25.5×, 8.9×, and 2.1× over SIGMA, SpArch, and SPADA, respectively. Second, SIGMA faces challenges in most workloads, despite utilizing a bitmap format for sparse matrix representation. Third, SpArch achieves a 2.9× speedup over SIGMA on average, which is attributed to the better input reuse of OutP execution flow and its efforts to reduce off-chip traffic during the merging phase. Fourth, SPADA, with its WA execution flow, provides an adaptive execution flow and better performance compared with accelerators with a fixed execution flow. On average, SPADA leads to a 4.2× speedup over SpArch and a 12.0× speedup over SIGMA.

Figure 8 presents a comparison of the total off-chip memory traffic for matrix **B** and partial outputs across various workloads. The memory traffic associated with matrix **B** is crucial for the effectiveness of multiplications. Similarly, efficient handling of reads and writes for partial outputs is vital for the merging processes. Figure 8 demonstrates that ACES incurs the lowest off-chip memory traffic compared to recent SpMM accelerators. On average, the off-chip memory traffic of SIGMA is 11.6× higher than that of ACES, while the traffic for SpArch and SPADA is 4.4× and 3.1× higher, respectively. SIGMA achieves efficient output reuse but faces significant challenges in input memory traffic. SpArch strives to reduce off-chip traffic during the merging phase through

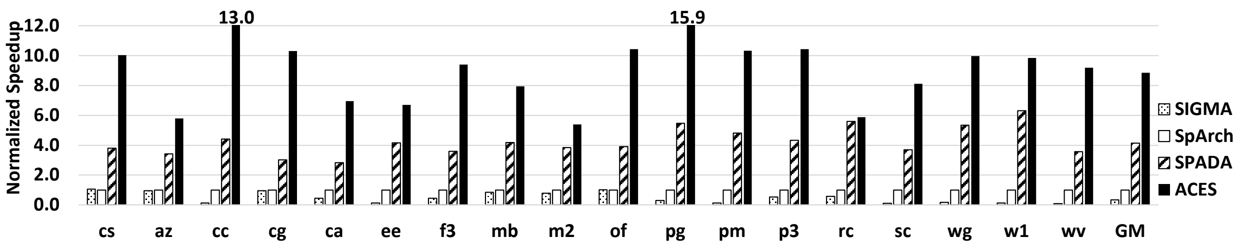

**Figure 7.** Performance comparison among SIGMA, SpArch, SPADA, and ACES.

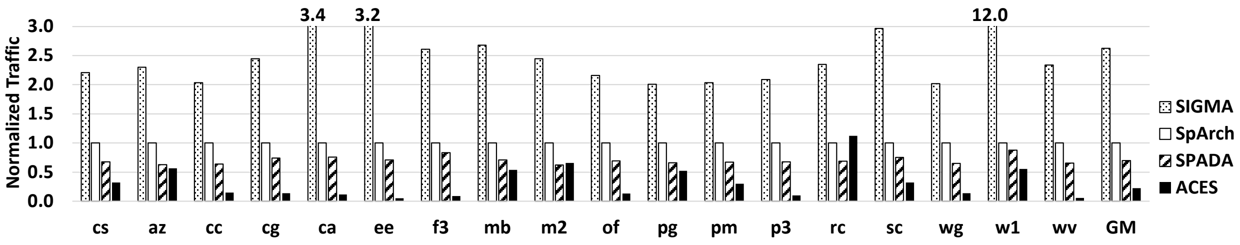

**Figure 8.** Memory traffic comparison among SIGMA, SpArch, SPADA, and ACES.

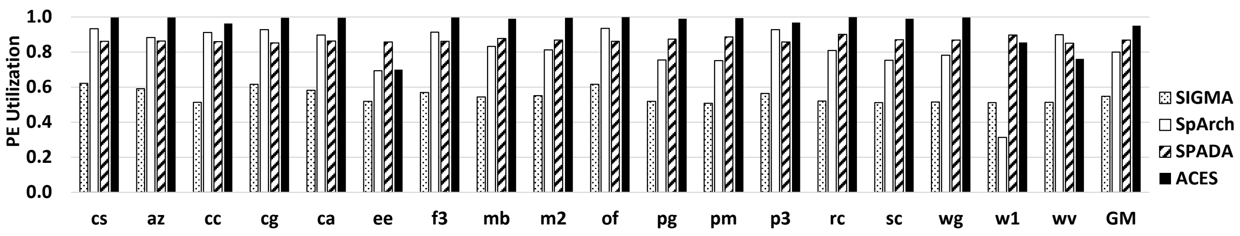

**Figure 9.** PE utilization comparison among SIGMA, SpArch, SPADA, and ACES.

an aggressive condensing representation of matrix **A**. However, it disrupts input reuse, leading to excessive fetching of different rows of matrix **B** and causing cache thrashing. SPADA, relying on its adaptive execution flow, reduces memory traffic compared with SIGMA and SpArch; however, it still exhibits a significant gap when compared with ACES. ACES, with its adaptive execution flow, effectively balances input and output reuse. The design of immediate merging, in conjunction with the PureFiber cache replacement policy, ensures finer granularity in merging partial results. Furthermore, the merging scheduler efficiently manages the final merging stage, collectively contributing to optimal traffic management.

Figure 9 compares PE utilization among four SpMM accelerators. On average, GAMMA, SpArch, SPADA, and ACES exhibit PE utilization rates of 54.8%, 80.0%, 86.9%, and 95.1%, respectively. Notably, ACES consistently maintains PE utilization rates above 90.0% in 15 of the 18 evaluated workloads. The high PE utilization of ACES is attributable to two main factors: First, its adaptive execution flow effectively mitigates synchronization risks. Second, the architecture of ACES features a one-to-one pairing of MPEs with APEs. This architecture, combined with efficient task scheduling by the synchronization scheduler during the immediate merging phase, reduces the collective dependency of MPEs and enhances the parallelism of APEs.

## 5.2 Performance with Different Condensing Degrees

To assess the effectiveness of the novel adaptive execution flow designed for ACES, we conduct a performance comparison across three static condensing degrees: none (No), aggressive (Ag), and moderate (Mo), complemented by adaptive condensing (Ad). For each condensing degree, we implement two distinct cache replacement policies: the traditional Least Recently Used (LRU) policy and the PureFiber (PF) policy. Figure 10 depicts the overall speedup of various ACES implementations over SpArch, highlighting the performance for each specific combination of condensing degrees and cache replacement policies, such as Ag-LRU (aggressive condensing with LRU policy) and Mo-PF (moderate condensing with PureFiber policy). Importantly, as the original ACES

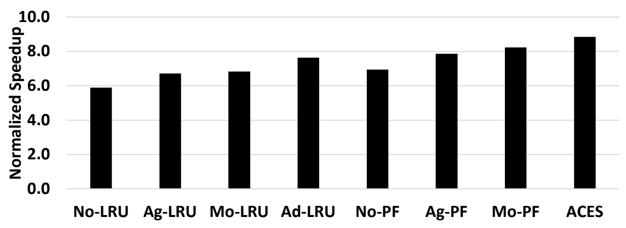

**Figure 10.** Average performance improvement of implementations with different static condensing degrees compared with adaptive condensing, across LRU and PureFiber cache replacement policies.

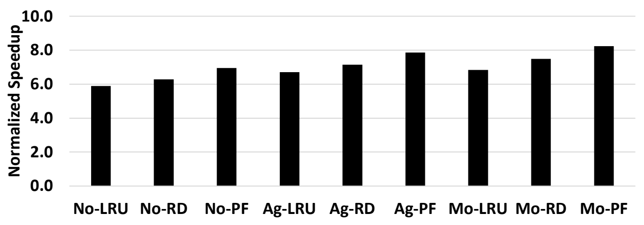

**Figure 11.** Average performance improvement of implementations with different cache replacement policies, across various static condensing degrees.

incorporates adaptive condensing with the PureFiber policy, the performance of the Ad-PF combination is distinctly identified as ACES in Figure 10.

The results presented in Figure 10 demonstrate the superior performance of adaptive condensing when compared with static condensing configurations under different cache replacement policies. Specifically, with the traditional LRU policy, adaptive condensing (Ad-LRU) achieves a 7.6× speedup over SpArch, outperforming No-LRU, Ag-LRU, and Mo-LRU by 29.6%, 13.8%, and 11.7%, respectively. Similarly, when utilizing the concurrency-aware PureFiber cache replacement policy, adaptive condensing (ACES) achieves an 8.9× speedup over SpArch, surpassing the No-PF, Ag-PF, and Mo-PF configurations by 27.4%, 12.6%, and 7.4%, respectively.

Adaptive condensing, whether working with the LRU or the PureFiber cache replacement policy, consistently delivers significant speedup gains over static condensing configurations. This performance advantage highlights the effectiveness of adaptive condensing in dynamically balancing data reuse and parallelism efficiency, providing an optimal execution flow finely tuned to the diverse sparse patterns of matrices. Such adaptability not only enhances the computational efficiency of ACES, but also establishes adaptive condensing as a pivotal feature for its versatility in diverse applications and system configurations. The ability of adaptive condensing to work effectively with various cache policies further underscores its broad applicability and flexibility in different system configurations.

### 5.3 Performance with Different Cache Replacement Policies

To highlight the innovation of the concurrency-aware cache replacement policy PureFiber in ACES, we compare it with two conventional cache replacement policies, LRU and RD, for managing the global cache. Specifically, the RD policy is designed to focus exclusively on the next request distance, prioritizing the eviction of cache lines that are furthest from being requested again. Both the LRU and RD policies aim to enhance data locality with the primary objective of minimizing cache misses. This comparative analysis is intended to showcase the distinctive advantages and potential of the concurrency-aware cache replacement strategy embodied by PureFiber. Unlike conventional policies that focus mainly on data locality, PureFiber considers both data locality and concurrency, potentially offering a more nuanced approach to cache management.

Figure 11 illustrates the overall speedup of various ACES implementations compared with SpArch, emphasizing the performance enhancements achieved with three distinct cache replacement policies: LRU, RD, and PureFiber. These policies are evaluated across three static condensing degrees: none, moderate, and aggressive, providing a comprehensive view of their impact on the performance of ACES. The key observation from Figure 11 is that across all static condensing degrees, the PureFiber cache replacement policy consistently leads to the highest speedups. Specifically, without condensing, the PureFiber policy (No-PF) achieves a speedup of 6.9×, which exceeds the performance of No-LRU and No-RD by 18.0% and 10.7%, respectively. With aggressive condensing, a 7.9× speedup of PureFiber (Ag-PF) is observed, surpassing Ag-LRU and Ag-RD by 17.2% and 10.0%, respectively. When utilizing the moderate condensing degree, PureFiber (Mo-PF) shows the most significant improvement with an 8.2× speedup, outperforming Mo-LRU and Mo-RD by 20.5% and 10.0%, respectively. These results indicate that utilizing the concurrency-aware PureFiber policy in cache replacement not only offers the best performance in the absence of condensing, but also provides superior results when working with both moderate and aggressive static condensing, underscoring its effectiveness.

Figure 12 displays the performance of ACES implementations with adaptive condensing across various cache replacement policies: LRU, RD, and PureFiber. Figure 12(a) illustrates the speedup achieved by the original ACES implementation employing the PureFiber policy, as well as the speedup of Ad-RD, both in comparison with Ad-LRU. We observe that, when working with adaptive condensing, the consideration of both data locality and concurrency enables ACES with the PureFiber policy to achieve an average improvement of 15.9% over the LRU baseline. In contrast, the average speedup for Ad-RD, which does not consider data concurrency, is 14.8%.

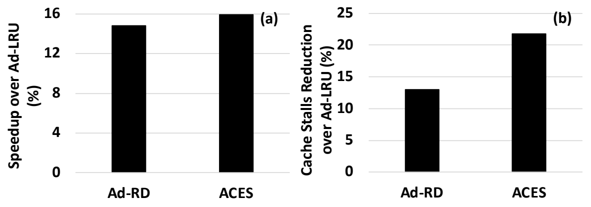

**Figure 12.** Average performance of implementations with adaptive condensing using different cache replacement policies, showcasing (a) speedup over Ad-LRU, and (b) reduction in cache stalls over Ad-LRU.

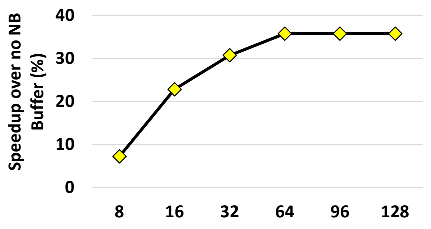

**Figure 13.** Performance of ACES across various NB buffer sizes.

Figure 12(b) offers a detailed analysis of the PureFiber performance. This figure highlights that the PureFiber policy effectively reduces cache stalls compared with other policies. Specifically, ACES with the PureFiber policy, achieves an average cache stall reduction of 21.8% over Ad-LRU, whereas Ad-RD, which considers only reuse distance, sees a cache stall reduction of 13.0%. These observations highlight that although reuse distance considerations can improve the global cache performance, the integration of concurrency awareness into the global cache management can yield additional benefits. By considering both data locality and concurrency, the PureFiber policy substantially diminishes cache stalls, thereby enhancing the overall efficiency of the cache system in SpMM accelerators like ACES.

## 5.4 Performance with Different NB Buffer Sizes

To showcase the effectiveness of integrating a NB buffer with the global cache in ACES, Figure 13 illustrates the overall speedup of ACES with varying sizes of the NB buffer, ranging from 8 to 64 subentries. Performance is normalized to a version of ACES that uses a blocking cache, without integrating the NB buffer with the global cache. Two key observations emerge from the results. First, integrating an NB buffer with the global cache significantly enhances performance. Even with a minimal 8-subentry NB buffer, ACES achieves a 7.2% performance improvement over the blocking baseline. Moreover, an ACES configuration with a 16-subentry NB buffer further achieves a 22.8% speedup. These results validate the critical role of the non-blocking cache in SpMM accelerator performance. The design of the NB buffer not only mitigates global cache stalls due to cache misses but also enables the global cache to support higher concurrency, thereby increasing parallelism. Second, Figure 13 reveals that the performance of ACES tends to stabilize once the number of subentries of the NB buffer reaches 64. At this point, a substantial performance improvement of 35.8% is observed. Additionally, it is important to consider that a larger NB buffer also introduces additional overhead. Therefore, in balancing performance gains with potential overheads, we set the default number of subentries in the NB buffer for ACES to 64.

**Table 3.** Area breakdown of ACES.

| Components | Area ($mm^2$) | Components | Area ($mm^2$) |
|---|---|---|---|
| 2 Fetchers | 0.22 | Global Buffer | 0.06 |
| 16 MPEs | 0.28 | Global Cache | 2.09 |
| 16 APEs | 0.24 | 16 SQs | 0.25 |
| 2 Schedulers | 0.14 | NB buffer | 0.08 |
| Crossbars | 0.16 | **Total** | **3.52** |

## 5.5 Area and Power

In ACES, the breakdown of the area is detailed in Table 3, showing a total area of 3.52mm$^2$. Similar to other SpMM accelerators, a significant portion of the area is occupied by the global cache. Specifically, the 1MB global cache in ACES constitutes 59.4% of the total area. Compared to existing works like SPADA and SpArch, the global cache capacity in ACES is relatively modest. By integrating a lightweight NB buffer with the global cache, ACES achieves reduced area overhead while improving performance relative to accelerators with larger caches. For comparison, the total areas of SPADA and SpArch at 28 nm technology are 6.32mm$^2$ and 13.96mm$^2$, respectively. Additionally, the power consumption evaluation of ACES reveals that it consumes a total of 2.83W of power.

## 6 Additional Related Works

**Accelerating SpMM on CPU and GPU.** Prior research has focused on accelerating SpMM on both CPU and GPU architectures [9, 10, 34, 41, 55]. MKL [55] offers math routines optimized for parallel computation using OpenMP on CPUs. For GPUs, cuSPARSE [41] enhances SpMM efficiency by parallelizing computations across matrix rows and using a hash table for merging partial results. CUSP [9] also adopts a parallel approach but employs a sorting algorithm for merging computations from different rows.

**SpMM Accelerators.** In this work, ACES is compared against three state-of-the-art SpMM accelerators: SIGMA [45], SpArch [61], and SPADA [32]. Additionally, a variety of other accelerators have been designed specifically to optimize SpMM [17, 22, 25, 39, 43, 50, 60]. Most of these accelerators adopt a fixed execution flow: for instance, ExTenor [22] utilizes an InP execution flow; OuterSPACE [43] and Spaghetti [25] adopt an OutP execution flow, while

GAMMA [60] and MatRaptor [50] employ a ROW execution flow. ACES distinguishes itself by dynamically adjusting its execution flow in response to the sparsity patterns of the input matrices. Moreover, ACES innovatively leverages concurrent data accesses to develop a cache replacement policy specifically designed for the global cache of an SpMM accelerator. These novel features, including the adaptive execution flow and concurrency-aware cache management, significantly accelerate SpMM processing.

**Sparse DNN Accelerators.** As the processing of diverse sparse matrices in DNNs becomes increasingly critical, the efficiency of sparse DNN accelerators [13, 18, 31, 39, 49, 62] grows more paramount. Flexagon [39], a configurable DNN accelerator, adapts its matrix multiplication execution flow to accommodate various DNN layers. It introduces a Merger-Reduction Network (MRN) to modify the execution flow across InP, OutP, and ROW based on offline analysis of matrix dimensions and sparsity patterns. CANDLES [18] adopts a channel-first architecture that traverses activations and weights to enhance the temporal locality of partial sums updated for an output neuron. It also incorporates a pixel-first compression method where matrices are segmented into groups for compression, storing non-zero values in tiles. Unlike Flexagon [39], which determines the execution flow through offline analysis, and CANDLES [18], which employs a static strategy, ACES dynamically adjusts its execution flow in response to the sparsity patterns of incoming matrices. This real-time adaptability allows ACES to balance parallel computing efficiency with data reuse, while simultaneously reducing overhead.

**Systolic Array-based Sparse DNN Accelerators.** The success of the Tensor Processing Unit (TPU) [27], which introduces the systolic array architecture, has inspired many recent systolic array-based DNN accelerators [11, 16, 21, 30, 56]. In order to enhance the utilization and computational efficiency of systolic arrays in handling matrices with diverse sparsity and sizes, column packing has been proposed. Kung et al. [30] address this by grouping the columns of a sparse matrix, combining multiple sparse columns into a single dense column, and mapping each group to a single column of the systolic array, significantly improving utilization efficiency. Inspired by [30], Sparse-TPU [21] introduces partition-wise packing, which divides the matrix into bands and packs columns within each band to minimize data conflicts. Moreover, Sparse-TPU employs a collision-aware algorithm to enhance the packing density, even in the presence of conflicts. While column packing significantly enhances the computational efficiency, it requires sparse matrix reordering and is performed offline, introducing extra pre-processing overhead. Conversely, ACES determines and adjusts adaptive condensing degrees dynamically during runtime, eliminating the need for pre-processing and the associated overheads.

**Concurrent Data Accesses.** Modern architectures now widely support data concurrency. For example, out-of-order execution [53] and simultaneous multithreading [54] have been instrumental in improving pipeline utilization. Additionally, advancements in memory and cache design like multi-port [63], pipelined [1], and non-blocking [28] caches allow more access to coexist within the same cycle, which substantially improves throughput and reduce memory access delays. Concurrent data accesses are utilized for performance modeling [40, 51, 52] and optimizations [35, 36, 38, 46, 58]. The C-AMAT model [52] enhances the traditional AMAT model [23] by quantitatively evaluating the combined impact of memory access locality and concurrency, accounting for data access overlaps. In the realm of cache management, the MLP-aware cache replacement policy [46] and the CARE framework [38] analyze the cost of each cache miss amid multiple concurrent outstanding accesses, guiding cache replacement decisions to reduce stalls and enhance performance. CHROME [35] offers a holistic solution by integrating cache replacement and bypassing with pattern-based prefetching, applying concurrency-aware system-level feedback to refine decision-making. In contrast to these concurrency-aware cache management studies [35, 37, 38, 46] that focus on general cache management enhancements, PureFiber is specifically designed for SpMM, leveraging the unique data access patterns of SpMM to optimize global cache usage in accelerators.

## 7 Conclusion

In this paper, we introduced ACES, an innovative SpMM accelerator. ACES supports an adaptive execution flow, adept at efficiently processing matrices with a wide range of sparse patterns. It also integrates co-optimizations of data locality and concurrency within its global cache, effectively reducing memory stalls and enhancing data access concurrency. The hardware architecture of ACES is meticulously tailored to complement its adaptive execution capabilities and cache optimizations, facilitating fine-granularity parallelism. Our comprehensive evaluations indicate that ACES consistently outperforms current state-of-the-art SpMM accelerators, underscoring its significant potential in enabling efficient computing solutions for diverse applications.

## Acknowledgments

We thank the anonymous reviewers for their helpful feedback. This research is supported in part by the National Science Foundation under Grants CNS-2310422, CNS-2152497, CCF-2029014, and CCF-2008907, and by the NSF-supported Chameleon testbed facility. Additional support comes from the National Key R&D Program of China (Grant 2018YFA0701500), the Strategic Priority Research Program of the Chinese Academy of Sciences (CAS) (Grant XDB44000000), the National Natural Science Foundation of China (Grants 62122076 and 62025404), and the Key Research Program of Frontier Sciences, CAS (Grant ZDBS-LY-JSC012).

## References

[1] Amit Agarwal, Kaushik Roy, and TN Vijaykumar. Exploring high bandwidth pipelined cache architecture for scaled technology. In *Proceedings of the conference on Design, Automation and Test in Europe-Volume 1*, page 10778. IEEE Computer Society, 2003.

[2] Mikhail Asiatici and Paolo Ienne. Stop crying over your cache miss rate: Handling efficiently thousands of outstanding misses in fpgas. In *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 310–319, 2019.

[3] Rajeev Balasubramonian, Andrew B Kahng, Naveen Muralimanohar, Ali Shafiee, and Vaishnav Srinivas. CACTI 7: New tools for interconnect exploration in innovative off-chip memories. *ACM Transactions on Architecture and Code Optimization (TACO)*, 14(2):1–25, 2017.

[4] Laszlo A. Belady. A study of replacement algorithms for a virtual-storage computer. *IBM Systems journal*, 5(2):78–101, 1966.

[5] Maciej Besta, Florian Marending, Edgar Solomonik, and Torsten Hoefler. Slimsell: A vectorizable graph representation for breadth-first search. In *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 32–41. IEEE, 2017.

[6] Aydin Buluç and Kamesh Madduri. Parallel breadth-first search on distributed memory systems. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–12, 2011.

[7] Andrew Canning, Giulia Galli, Francesco Mauri, Alessandro De Vita, and Roberto Car. O (n) tight-binding molecular dynamics on massively parallel computers: an orbital decomposition approach. *Computer Physics Communications*, 94(2-3):89–102, 1996.

[8] Timothy M Chan. More algorithms for all-pairs shortest paths in weighted graphs. In *Proceedings of the thirty-ninth annual ACM symposium on Theory of computing*, pages 590–598, 2007.

[9] Steven Dalton, Nathan Bell, Luke Olson, and Michael Garland. Cusp: Generic parallel algorithms for sparse matrix and graph computations, 2014.

[10] Steven Dalton, Luke Olson, and Nathan Bell. Optimizing sparse matrix—matrix multiplication for the gpu. *ACM Transactions on Mathematical Software (TOMS)*, 41(4):1–20, 2015.

[11] Saptarsi Das, Arnab Roy, Kiran Kolar Chandrasekharan, Ankur Deshwal, and Sehwan Lee. A systolic dataflow based accelerator for CNNs. In *2020 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 1–5. IEEE, 2020.

[12] Timothy A Davis and Yifan Hu. The University of Florida sparse matrix collection. *ACM Transactions on Mathematical Software (TOMS)*, 38(1):1–25, 2011.

[13] Chunhua Deng, Siyu Liao, Yi Xie, Keshab K Parhi, Xuehai Qian, and Bo Yuan. PermDNN: Efficient compressed DNN architecture with permuted diagonal matrices. In *2018 51st Annual IEEE/ACM international symposium on microarchitecture (MICRO)*, pages 189–202. IEEE, 2018.

[14] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

[15] Jianhua Gao, Weixing Ji, Fangli Chang, Shiyu Han, Bingxin Wei, Zeming Liu, and Yizhuo Wang. A systematic survey of general sparse matrix-matrix multiplication. *ACM Computing Surveys*, 55(12):1–36, 2023.

[16] Hasan Genc, Seah Kim, Alon Amid, Ameer Haj-Ali, Vighnesh Iyer, Pranav Prakash, Jerry Zhao, Daniel Grubb, Harrison Liew, Howard Mao, et al. Gemmini: Enabling systematic deep-learning architecture evaluation via full-stack integration. In *2021 58th ACM/IEEE Design Automation Conference (DAC)*, pages 769–774. IEEE, 2021.

[17] Gerasimos Gerogiannis, Serif Yesil, Damitha Lenadora, Dingyuan Cao, Charith Mendis, and Josep Torrellas. SPADE: A Flexible and Scalable Accelerator for SpMM and SDDMM. In *Proceedings of the 50th Annual International Symposium on Computer Architecture*, pages 1–15, 2023.

[18] Sumanth Gudaparthi, Sarabjeet Singh, Surya Narayanan, Rajeev Balasubramonian, and Visvesh Sathe. CANDLES: Channel-aware novel dataflow-microarchitecture co-design for low energy sparse neural network acceleration. In *2022 IEEE International Symposium on high-performance computer architecture (HPCA)*, pages 876–891. IEEE, 2022.

[19] Song Han, Xingyu Liu, Huizi Mao, Jing Pu, Ardavan Pedram, Mark A Horowitz, and William J Dally. EIE: Efficient inference engine on compressed deep neural network. *ACM SIGARCH Computer Architecture News*, 44(3):243–254, 2016.

[20] Song Han, Huizi Mao, and William J Dally. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. *arXiv preprint arXiv:1510.00149*, 2015.

[21] Xin He, Subhankar Pal, Aporva Amarnath, Siying Feng, Dong-Hyeon Park, Austin Rovinski, Haojie Ye, Yuhan Chen, Ronald Dreslinski, and Trevor Mudge. Sparse-TPU: Adapting systolic arrays for sparse matrices. In *Proceedings of the 34th ACM international conference on supercomputing*, pages 1–12, 2020.

[22] Kartik Hegde, Hadi Asghari-Moghaddam, Michael Pellauer, Neal Crago, Aamer Jaleel, Edgar Solomonik, Joel Emer, and Christopher W Fletcher. Extensor: An accelerator for sparse tensor algebra. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 319–333, 2019.

[23] John L Hennessy and David A Patterson. *Computer Architecture: A Quantitative Approach.* Elsevier, 2019.

[24] Torsten Hoefler and Marc Snir. Generic topology mapping strategies for large-scale parallel architectures. In *Proceedings of the international conference on Supercomputing*, pages 75–84, 2011.

[25] Reza Hojabr, Ali Sedaghati, Amirali Sharifian, Ahmad Khonsari, and Arrvindh Shriraman. Spaghetti: Streaming accelerators for highly sparse gemm on fpgas. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 84–96. IEEE, 2021.

[26] David A Huffman. A method for the construction of minimum-redundancy codes. *Proceedings of the IRE*, 40(9):1098–1101, 1952.

[27] Norman P Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, et al. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th annual international symposium on computer architecture*, pages 1–12, 2017.

[28] David Kroft. Lockup-free instruction fetch/prefetch cache organization. In *Proceedings of the 8th annual symposium on Computer Architecture*, pages 81–87. IEEE Computer Society Press, 1981.

[29] David Kroft. Lockup-free instruction fetch/prefetch cache organization. In *25 years of the international symposia on Computer architecture (selected papers)*, pages 195–201, 1998.

[30] HT Kung, Bradley McDanel, and Sai Qian Zhang. Packing sparse convolutional neural networks for efficient systolic array implementations: Column combining under joint optimization. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 821–834, 2019.

[31] Shuangchen Li, Dimin Niu, Krishna T Malladi, Hongzhong Zheng, Bob Brennan, and Yuan Xie. Drisa: A dram-based reconfigurable in-situ accelerator. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 288–301, 2017.

[32] Zhiyao Li, Jiaxiang Li, Taijie Chen, Dimin Niu, Hongzhong Zheng, Yuan Xie, and Mingyu Gao. Spada: Accelerating Sparse Matrix Multiplication with Adaptive Dataflow. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, pages 747–761, 2023.

[33] Baoyuan Liu, Min Wang, Hassan Foroosh, Marshall Tappen, and Marianna Pensky. Sparse convolutional neural networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 806–814, 2015.

[34] Weifeng Liu and Brian Vinter. An efficient GPU general sparse matrix-matrix multiplication for irregular data. In *2014 IEEE 28th International*

*Parallel and Distributed Processing Symposium*, pages 370–381. IEEE, 2014.

[35] Xiaoyang Lu, Hamed Najafi, Jason Liu, and Xian-He Sun. CHROME: Concurrency-aware holistic cache management framework with online reinforcement learning. In *2024 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 2024.

[36] Xiaoyang Lu, Rujia Wang, and Xian-He Sun. Apac: An accurate and adaptive prefetch framework with concurrent memory access analysis. In *2020 IEEE 38th International Conference on Computer Design (ICCD)*, pages 222–229. IEEE, 2020.

[37] Xiaoyang Lu, Rujia Wang, and Xian-He Sun. Premier: A concurrency-aware pseudo-partitioning framework for shared last-level cache. In *2021 IEEE 39th International Conference on Computer Design (ICCD)*, pages 391–394. IEEE, 2021.

[38] Xiaoyang Lu, Rujia Wang, and Xian-He Sun. CARE: A concurrency-aware enhanced lightweight cache management framework. In *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 1208–1220. IEEE, 2023.

[39] Francisco Muñoz-Martínez, Raveesh Garg, Michael Pellauer, José L Abellán, Manuel E Acacio, and Tushar Krishna. Flexagon: A Multi-Dataflow Sparse-Sparse Matrix Multiplication Accelerator for Efficient DNN Processing. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, pages 252–265, 2023.

[40] Hamed Najafi, Jason Liu, Xiaoyang Lu, and Xian-He Sun. A generalized model for modern hierarchical memory system. In *2022 Winter Simulation Conference (WSC)*, pages 2178–2188. IEEE, 2022.

[41] Maxim Naumov, L Chien, Philippe Vandermersch, and Ujval Kapasi. Cusparse library. In *GPU Technology Conference*, 2010.

[42] Maxim Naumov, Dheevatsa Mudigere, Hao-Jun Michael Shi, Jianyu Huang, Narayanan Sundaraman, Jongsoo Park, Xiaodong Wang, Udit Gupta, Carole-Jean Wu, Alisson G Azzolini, et al. Deep learning recommendation model for personalization and recommendation systems. *arXiv preprint arXiv:1906.00091*, 2019.

[43] Subhankar Pal, Jonathan Beaumont, Dong-Hyeon Park, Aporva Amarnath, Siying Feng, Chaitali Chakrabarti, Hun-Seok Kim, David Blaauw, Trevor Mudge, and Ronald Dreslinski. Outerspace: An outer product based sparse matrix multiplication accelerator. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 724–736. IEEE, 2018.

[44] Cosmin G Petra, Olaf Schenk, Miles Lubin, and Klaus Gärtner. An augmented incomplete factorization approach for computing the Schur complement in stochastic optimization. *SIAM Journal on Scientific Computing*, 36(2):C139–C162, 2014.

[45] Eric Qin, Ananda Samajdar, Hyoukjun Kwon, Vineet Nadella, Sudarshan Srinivasan, Dipankar Das, Bharat Kaul, and Tushar Krishna. Sigma: A sparse and irregular gemm accelerator with flexible interconnects for dnn training. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 58–70. IEEE, 2020.

[46] Moinuddin K Qureshi, Daniel N Lynch, Onur Mutlu, and Yale N Patt. A case for MLP-aware cache replacement. *ACM SIGARCH Computer Architecture News*, 34(2):167–178, 2006.

[47] Raimund Seidel. On the all-pairs-shortest-path problem in unweighted undirected graphs. *Journal of computer and system sciences*, 51(3):400–403, 1995.

[48] Korey Sewell, Ronald G Dreslinski, Thomas Manville, Sudhir Satpathy, Nathaniel Pinckney, Geoffrey Blake, Michael Cieslak, Reetuparna Das, Thomas F Wenisch, Dennis Sylvester, et al. Swizzle-switch networks for many-core systems. *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, 2(2):278–294, 2012.

[49] Yakun Sophia Shao, Jason Clemons, Rangharajan Venkatesan, Brian Zimmer, Matthew Fojtik, Nan Jiang, Ben Keller, Alicia Klinefelter, Nathaniel Pinckney, Priyanka Raina, et al. Simba: Scaling deep-learning inference with multi-chip-module-based architecture. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 14–27, 2019.

[50] Nitish Srivastava, Hanchen Jin, Jie Liu, David Albonesi, and Zhiru Zhang. Matraptor: A sparse-sparse matrix multiplication accelerator based on row-wise product. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 766–780. IEEE, 2020.

[51] Xian-He Sun and Xiaoyang Lu. The Memory-Bounded Speedup Model and Its Impacts in Computing. *Journal of Computer Science and Technology*, 38(1):64–79, 2023.

[52] Xian-He Sun and Dawei Wang. Concurrent average memory access time. *Computer*, 47(5):74–80, 2013.

[53] Robert M Tomasulo. An efficient algorithm for exploiting multiple arithmetic units. *IBM Journal of research and Development*, 11(1):25–33, 1967.

[54] Dean M Tullsen, Susan J Eggers, and Henry M Levy. Simultaneous multithreading: Maximizing on-chip parallelism. In *ACM SIGARCH Computer Architecture News*, volume 23, pages 392–403. ACM, 1995.

[55] Endong Wang, Qing Zhang, Bo Shen, Guangyong Zhang, Xiaowei Lu, Qing Wu, and Yajuan Wang. High-performance computing on the intel xeon phi. *Springer*, 5:2, 2014.

[56] Rui Xu, Sheng Ma, Yaohua Wang, Xinhai Chen, and Yang Guo. Configurable multi-directional systolic array architecture for convolutional neural networks. *ACM Transactions on Architecture and Code Optimization (TACO)*, 18(4):1–24, 2021.

[57] Ichitaro Yamazaki and Xiaoye S Li. On techniques to improve robustness and scalability of a parallel hybrid linear solver. In *International Conference on High Performance Computing for Computational Science*, pages 421–434. Springer, 2010.

[58] Liang Yan, Mingzhe Zhang, Rujia Wang, Xiaoming Chen, Xingqi Zou, Xiaoyang Lu, Yinhe Han, and Xian-He Sun. Copim: a concurrency-aware pim workload offloading architecture for graph applications. In *2021 IEEE/ACM International Symposium on Low Power Electronics and Design (ISLPED)*, pages 1–6. IEEE, 2021.

[59] Raphael Yuster and Uri Zwick. Detecting short directed cycles using rectangular matrix multiplication and dynamic programming. In *SODA*, volume 4, pages 254–260. Citeseer, 2004.

[60] Guowei Zhang, Nithya Attaluri, Joel S Emer, and Daniel Sanchez. Gamma: Leveraging Gustavson's algorithm to accelerate sparse matrix multiplication. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 687–701, 2021.

[61] Zhekai Zhang, Hanrui Wang, Song Han, and William J Dally. Sparch: Efficient architecture for sparse matrix multiplication. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 261–274. IEEE, 2020.

[62] Xuda Zhou, Zidong Du, Qi Guo, Shaoli Liu, Chengsi Liu, Chao Wang, Xuehai Zhou, Ling Li, Tianshi Chen, and Yunji Chen. Cambricon-S: Addressing irregularity in sparse neural networks through a cooperative software/hardware approach. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 15–28. IEEE, 2018.

[63] Zhaomin Zhu, Koh Johguchi, Hans Jürgen Mattausch, Tetsushi Koide, Tai Hirakawa, and Tetsuo Hironaka. A novel hierarchical multi-port cache. In *Solid-State Circuits Conference, 2003. ESSCIRC'03. Proceedings of the 29th European*, pages 405–408. IEEE, 2003.