La Superba: Leveraging a Self-Comparison Method to Understand the Performance Benefits of Sparse Acceleration Optimizations

Nebil Ozer[1], Gregory Kollmer[2*,3], Ramyad Hadidi[2*], and Bahar Asgari[1]

[1]*University of Maryland, College Park*
[2]*Rain AI*
[3]*University of Texas, Austin*

***Abstract*—Evaluation of modern sparse accelerators is essential for enhancing their performance and guiding future research. However, modern fine-grained assessments of individual optimizations within these accelerators remain challenging and often lead to inconsistencies between proposals. To address this issue, we introduce La Superba, an optimization-granular analysis of prior studies using a self-comparison technique. La Superba expands existing approaches for analyzing constituent parts of sparse accelerators and introduces a set of categorizations for these optimizations. Our method involves isolating each optimization by systematically removing it from the accelerator and comparing the modified accelerator to its original configuration. This technique is applied to optimizations within three state-of-the-art sparse matrix multiplication accelerators, identifying and examining their individual impacts as well as categorizing them to uncover broader patterns in the performance benefits they offer. Through targeted case studies, we demonstrate the performance effects of some of our chosen optimizations, providing deeper insights into their contributions than those detailed in the original studies. We conclude the paper by discussing overarching patterns in optimization structures, utilizing our categorization as a foundation. By establishing a structured framework for evaluation, this work not only clarifies the contributions of specific optimizations but also provides a valuable tool for future designers and researchers to systematically assess and innovate within the sparse accelerator design space.**

I. Introduction

Sparse-sparse matrix multiplication (SpGEMM) is a key component in various application domains such as scientific computing [1]–[4], machine learning [5]–[14], and big data analytics [15]–[18]. Executing SpGEMM remains challenging [8], [19]–[21] due to the inherent irregularity and complexity of sparse matrices, which require efficient handling of non-zero elements and can lead to computational and memory access inefficiencies despite compression techniques [7], [22], [23]. Specialized sparse accelerators [6], [19], [24]–[30] have been developed to address these inefficiencies. SpGEMM accelerators are typically characterized by a set of design choices (optimizations) that give the accelerator an advantage when operating on sparse datasets [5]–[7], [25], [31]. However, current evaluation methods for optimizations [7], [31], [32] frequently evaluate them in non-standard ways that do not necessarily translate across works. Such evaluations can overlook important insights hidden in individual optimizations and make them difficult to organize and compare across accelerators and implementations.

The high-level research question addressed in this paper is: "How can we develop quality evaluations of optimizations that are meaningful, valuable, and simple?" Our work, La Superba[‡], employs a self-comparison methodology to dissect and evaluate individual optimizations within sparse accelerators, enabling a deep understanding of how each design decision contributes to performance improvements in SpGEMM workloads while still allowing for evaluation and categorization of optimizations across accelerators. By expanding existing evaluation methods [5], [6], [8], [33]–[35], our approach provides a framework to assess the performance benefits of various SpGEMM optimizations across diverse datasets, accelerators, and optimization types. Our findings demonstrate that the behavior of an optimization can frequently be tied to what part of the accelerator it optimizes. For example, optimizations that focus on computation become more critical as inputs become denser and the accelerator becomes compute-bound. We also found that certain optimizations, initially perceived as impactful, can fail to bring performance improvements based on input type, highlighting the importance of holistic evaluation methods.

The key contributions of this work are as follows: (1) We introduce a method that measures the impact a specific optimization has on the overall performance of the accelerator. (2) We introduce a simple categorization framework to organize and evaluate optimizations by what areas of the accelerator they target. (3) We identify a set of "case study" optimizations within recent research where our method of evaluation could add valuable context. (4) We evaluate these categories and case studies using our methodology and showcase how our evaluation adds meaningful insight about general trends among the optimizations in the targeted papers.

[*]This work was done independently of employment with Rain AI

[‡]La Superba is a red giant star in the constellation Canes Venatici.

II. Why *La Superba*?

New accelerator designs frequently incorporate optimizations from existing designs, leveraging proven strategies to enhance performance [6], [7], [24]. By categorizing and evaluating these strategies on a smaller scale – making them more accessible and practical for researchers to integrate into their own designs – we provide an evaluation that is easier to digest and apply. This work not only helps researchers understand which optimizations contribute most significantly to performance but also clarifies the specific ways they do so. Gaining this understanding is crucial, as it offers valuable insights that can inform design decisions and highlight areas for further refinement or innovation [13], [36]–[38].

Building on the insights gained from evaluating these optimizations individually, we categorize and sort them into groups based on the areas they enhance, determining the performance metrics affected and the range of improvements produced by each category, and reporting on general patterns across a wide population of optimizations. By sorting optimizations into distinct categories, we can determine which types of optimizations introduced in these works result in the greatest performance improvements and conclude why some categories are more effective for enhancing performance on specific input types. This categorization also serves as an introduction to a modular approach for these accelerators, allowing researchers to identify trade-offs between optimizations within the same category, as well as synergies with optimizations in other categories. This refined understanding of the efficacy of each optimization category ultimately aids in developing more targeted and efficient sparse matrix multiplication accelerators, highlighting which categories have historically contributed the most to speedup, in what conditions this speedup is maximized, and the extent to which these optimizations contribute to the overall work.

III. Background & Methodology

This section first introduces the three common matrix multiplication strategies used in SpGEMM accelerators, followed by a breakdown of the three optimization categories that our work introduces. We then provide an overview of all of our optimizations and introduce our case study optimizations – specific optimizations we select for a more extensive review.

A. Matrix Multiplication Strategies

Most SpGEMM accelerators use one of three matrix multiplication strategies, as presented in Figure 1. **Inner product** [31], [39] computes the dot product of rows and columns, iterating through one row of the *A matrix* and one column of the *B matrix* to accumulate a single scalar value of the *C matrix*. This strategy is primarily used in human calculation of matrix multiplication, as it does not require the saving and summation of partial outputs. Despite this, inner product is poorly suited for SpGEMM computations due to its redundant input fetches [6], [40], which has resulted in a lack of inner product SpGEMM accelerators. **Outer product** [6], [19], [26] on the other hand, iterates through one column of the *A matrix* and one row of the *B matrix* to produce a partial result spanning the entire *C matrix*. All partial matrices are then merged to obtain the final *C matrix*. In this context, merging refers to combining streams of ordered (coordinate, data) pairs by combining them into an output stream in order of coordinate, summing values with shared coordinates. **Row-wise product** [5], [7], [41] iterates through rows of the *A matrix* and multiplies each value with a corresponding row of the *B matrix*. The output is a partial row of the *C matrix*, with the row coordinate taken from the *A matrix* and the column coordinates from the *B matrix*. When all partial rows sharing the same row number are merged, a row of the *C matrix* is obtained.

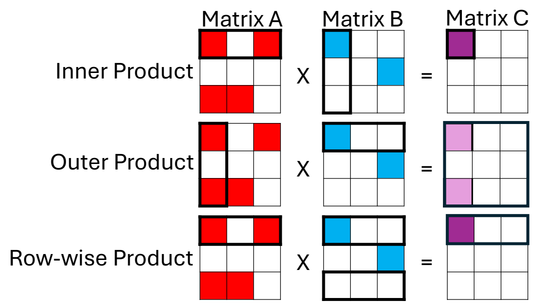
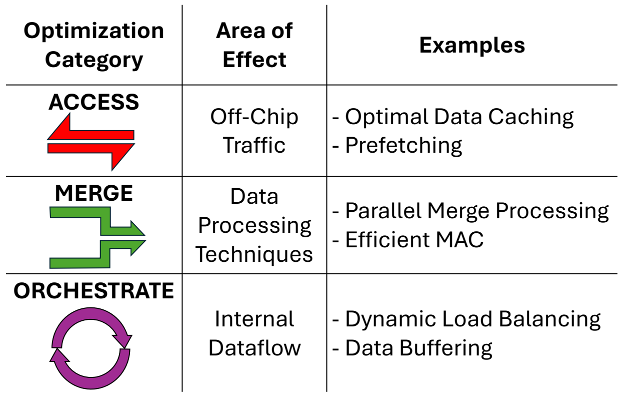


Fig. 1: Overview of the three common matrix-matrix multiplication strategies used in SpGEMM accelerators. Thick borders highlight the rows or columns that each strategy's outermost iteration processes to generate corresponding output or partial output elements.

B. Optimization Categories

Since each matrix multiplication strategy requires vastly different dataflows, optimizations come in all shapes and sizes and operate in largely unique ways [5]–[7], [24], [27], [31]. Despite this, we posit that nearly every dataflow optimization found in modern SpGEMM accelerators can be broadly categorized into a(n) **Access**, **Merge**, or **Orchestrate** optimization by what purpose it serves within the accelerator's



Optimization Category	Area of Effect	Examples
ACCESS	Off-Chip Traffic	- Optimal Data Caching - Prefetching
MERGE	Data Processing Techniques	- Parallel Merge Processing - Efficient MAC
ORCHESTRATE	Internal Dataflow	- Dynamic Load Balancing - Data Buffering

Fig. 2: The three categories of optimizations used in La Superba and their corresponding area of effect and examples.

TABLE I: List of optimizations, their categories, what accelerator they stem from (Gamma [5], MatRaptor [7], SpArch [6]) and a description of what they do. Optimizations 3, 7, and 9 are the case studies.

#	Optimization	Category	Accelerator/Strategy	Description
1.	Affinity-Based Reordering	Access	Gamma/Row-Wise	Re-orders rows to improve cache reuse
2.	FiberCache	Access	Gamma/Row-Wise	Optimal cache replacement using priority system
3.	**C2SR Compression**	Access	MatRaptor/Row-Wise	Custom compression format to minimize memory waste
4.	Lookahead Cache	Access	SpArch/Outer	Spills rows with low temporal locality using a lookahead FIFO
5.	High-Radix Merger	Merge	Gamma/Row-Wise	Uses a tree of compute units to merge many inputs at a time
6.	Multi-Queue Merging	Merge	MatRaptor/Row-Wise	Reduces merge latency by shuffling values between Queues
7.	**Array-Based Merger**	Merge	SpArch/Outer	Uses Comparator array to improve merge throughput
8.	A-Matrix Compression	Merge	SpArch/Outer	"Pre-merges" partial matrices by merging input columns
9.	**Dynamic Row Scheduler**	Orchestrate	Gamma/Row-Wise	Maximizes cache efficiency by parallelizing work on output rows
10.	Dual-Phase PE	Orchestrate	MatRaptor/Row-Wise	PEs work on input and output in two decoupled phases.
11.	SpAL/SpBl FIFOs	Orchestrate	MatRaptor/Row-Wise	Inputs between matrix loaders are buffered to reduce latency
12.	Merge Tree FIFOs	Orchestrate	SpArch/Outer	Stops bottlenecks by buffering values within merge tree

dataflow. The relationship between the three optimization categories is visualized in Figure 2. Note that these categories do not indicate where in the accelerator the optimization occurs, but rather the purpose of the optimization, and what aspect of the dataflow it aims to improve. *Access* optimizations encompass techniques that optimize dataflow by affecting off-chip memory accesses. Though they can take many forms, these optimizations can generally be identified through lower DRAM latency or a reduction in the number of memory accesses.

Merge optimizations focus on improving the often costly operations of multiplying/accumulating (MAC) and merging values within the accelerator. Finally, *Orchestrate* is a category for optimizations targeting internal dataflow; alleviating bottlenecks on-chip, ensuring coordination, balance, and high utilization within and between hardware components. These categories nearly always emerge in SpGEMM accelerators in one form or another, since the majority of the computation the accelerator does is encompassed by one of these three categories. For example, Merge optimizations often show up when merging partial outputs, such as the partial rows generated in row-wise product. At a high level, **Access** optimizes off-chip data movement, **Merge** optimizes accumulation and merging of data, and **Orchestrate** optimizes internal data movement and workload balance.

C. Introducing Our Optimizations

To investigate our optimization categories, we chose to examine the twelve optimizations listed in Table I. In our selection and analysis, for the sake of uniformity, we focus on three recent proposed architectures targeting ASIC implementation: MatRaptor [7], Gamma [5], and SpArch [6]. Additionally, we select optimizations that lend themselves to being isolated and measured, a counterexample of which would be near-memory processing, which introduces additional layers of complexity that are difficult to pick apart and measure. Finally, we focus only on optimizations that have a meaningful impact on the accelerator. For example, a multiplier array to speed up multiplication throughput would be considered effectual or meaningful. On the other hand, small data buffers in a noncritical section of the accelerator are important in their own way, but offer little value for study and are largely unimportant in the context of the accelerator as a whole.

TABLE II: The parameter ranges for the 30 Square Matrices from SuiteSparse [42].

Sparsity	40% (mbeacxc) - 0.0006% (web-Google)
nonzeros	66 (bcsstm) - 9,080,404 (gearbox)
Row & Column Length	142 (bcsstm) - 916,428 (web-Google)
nonzeros / Row	0.5 (bcsstm) - 100.6 (mbeacxc)

TABLE III: Parameters used to generate our two synthetic datasets. Matrices are square, so size is reported as a single side length. S = matrix size, D (%) = density as a percentage

Dataset	Const Param	Variable Params
Sparsity	S - 10000	D (%) - 1, 0.5, 0.1, 0.05, 0.01, 0.005, 0.001
Size	D (%) - 0.1	S - 50, 100, 500, 1000, 5000, 10000, 50000

In addition to evaluating the broad dataflow optimization categories, we select a set of three **Case Study Optimizations** to analyze at a granular level – optimizations number 3, 7, and 9 in Table I. These optimizations fulfill two additional requirements: First, evaluating the optimization must provide some important insight that was unclear or not expanded on in the original work. Second, the optimization must be one of the core contributions of the prior work. This means that it must be one of the primary optimizations that the work presents, often being listed as a notable part of the accelerator within the first few sections, and having a dedicated section within the work to explain its function. Section V explains the case-study optimizations in detail.

IV. Experimental Setup

This section describes the metrics, datasets, data analysis methodology, simulation set up, and validation methodology used in our studies.

A. Metrics & Datasets

Throughout our experiments, we measure several key performance metrics. These metrics include the runtime, total memory accesses, memory bandwidth utilization, and cache hit/miss rate of the entire accelerator (if applicable), as well

TABLE IV: Hardware configurations for the three accelerator simulators.

(a) MatRaptor

Configuration	Value
PE Array Size	8
Number of Sorting Queues	20 (2 × 10)
Sorting Queue Sizes	4 KB
Inter-Component FIFO Sizes	0.6 KB
Memory Channels	8

(b) Gamma

Configuration	Value
Number of PEs	32
PE Radix	64
FiberCache Size	3MB
FiberCache Banks	48
Memory Channels	8

(c) SpArch

Configuration	Value
Merge Tree Size	6 layers, 64 inputs
Merge Tree FIFO Size	256 Bytes
Array Merger Size	16 × 16
Lookahead FIFO Size	32KB
Prefetch Buffer Size	0.5MB
Memory Channels	16

as the hardware utilization and throughput of individual components. These measurements are taken over three distinct sets of data. The first set consists of thirty real-world sparse matrices sourced from SuiteSparse [42], as summarized in Table II. These data points are meant to measure the average real-world performance of the targeted accelerators with and without each optimization.

The synthetic workloads consist of two datasets, detailed in Table III. The first dataset measures how each optimization interacts with input sparsity, consisting of 7 synthetic matrices with a fixed dimensionality of 10,000 × 10,000 and varying densities from 1% down to 0.001%. The second dataset explores the effect of matrix size, maintaining a constant sparsity of 0.1% while varying dimensions from 50 × 50 up to 50,000 × 50,000. In both datasets, each matrix is expected to be evaluated individually to show distinct data points rather than aggregates.

Given that this study relies heavily on self-comparison, most of the data is normalized. We dive deeper into our implementations in the next subsection, but as a general rule we report performance metrics of implementations without our targeted optimization normalized to the performance metrics of its fully optimized implementation. Each data point is reported as a ratio, e.g., "Runtime for Merge optimizations normalized to baseline".

B. Simulation

For our experiments, we developed a set of three custom cycle-accurate C++ simulators – one for each accelerator – rather than relying on released simulators or off-the-shelf tools. This decision stems from two key considerations: (1) the need to precisely control and exclude specific optimizations for our self-comparison methodology, and (2) the lack of publicly available simulators that could match our requirements for accuracy, fidelity, and flexibility. By implementing these simulators ourselves, we ensure that the results are directly comparable, as each implementation operates under the same controlled environment and modeling assumptions. Each simulator has five versions: a baseline implementation, which replicates the original accelerator design, and four alternative implementations, each omitting one of the optimizations specified in Table I.

The simulators are configured to match the configurations stated to be optimal in the corresponding prior works. This includes aspects such as PE size, cache size, and bandwidth limitations, detailed in Table IV. We omit full architectural details for brevity, but have included relevant hardware configuration information in each case study.

Each accelerator is simulated as a series of components, each taking input through a register, performing an operation each cycle, and saving output to another register. The simulators operate at a modeled clock frequency of 1 GHz, with one-cycle delays enforced between components to account for the time needed to access values placed in input registers.

While it is reasonable to develop a cycle-accurate simulator for straightforward on-chip components and interactions, the high levels of complexity involved in modern DRAM systems, especially in implementations such as HBM, were unreasonable and unnecessary to simulate ourselves. We use the Ramulator 2.0 [43] DRAM simulator to accurately measure memory access timing, motivated by its flexibility and extensibility for modeling diverse DRAM configurations. A wrapper was built around the DRAM timer, capable of receiving and forwarding requests from the simulators to Ramulator, and informing the simulators when memory requests are fulfilled.

Our primary memory configuration consists of 8 HBM memory channels, each with 128-bit buses. For SpArch, which uses 16 64-bit buses, we adjust Ramulator's configuration to align with its setup. Each memory channel is divided into 2 pseudo-channels – except for SpArch [6], which uses 1 pseudo-channel per channel – and has a burst length of 4, resulting in 32 bytes of data being returned per memory access [44]. Ramulator operates at a clock speed of 1 GHz with a data rate of 2 GT/s, providing a theoretical peak bandwidth of 256 GB/s across all memory channels [43]. Note that the above will not be changed in any of our experiments.

C. Validation

To ensure the simulators are reliable, we validated them thoroughly at both the component and system levels. Each simulator component was tested individually to confirm cycle-accurate behavior. Initially, we defined a set of specifications for each component, using descriptions from the prior work whenever possible and external references [45]–[47] when details were missing. These specifications primarily outlined the resources a specific action would take (runtime, memory, space, etc.), and we built and tested each component to align with them. For standard hardware components such as caches, FIFOs, and control logic, we followed established standards to guarantee accuracy. Without direct access to simulators or hardware descriptions from the prior work, this approach

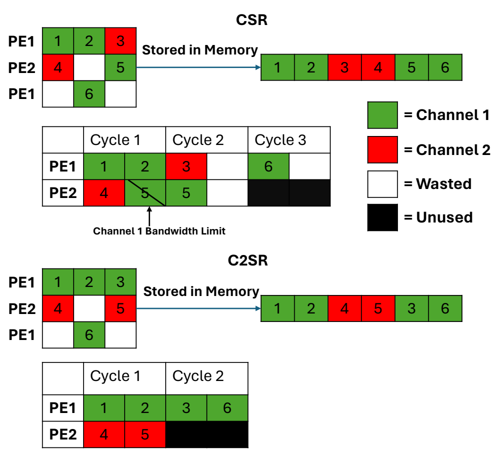


Fig. 3: Example demonstrating how C2SR prevents wasted memory cycles through channel interleaving.

represents the best available method to ensure fine-grained correctness.

In addition to individual component testing, we performed extensive end-to-end validation to confirm overall system correctness. These tests involved simulations using dense and sparse input matrices up to $20,000 \times 20,000$ in size. To verify simulator accuracy, we compared results computed by the simulators directly against trusted calculations from NumPy, a reliable numerical computing library. These comparisons confirmed that the simulators produce correct results and perform accurately even when under stress.

V. Case Studies

In our case studies, we first introduce each case study optimization and give some context as well as an overview of its purpose. We then introduce what results the prior work showed about the optimization, and where there is room to further investigate the optimization. Finally, we report results from the simulators and describe what benefits our evaluation brings to researchers.

We have included a high-level overview of the accelerator structure in each of the Case Studies. Unless change is required for testing a specific optimization, each of the experiments implement the same hardware configurations located in Table IV.

A. Case Study 1: C2SR Compression

C2SR is an **Access** optimization introduced in MatRaptor [7]. C2SR is used to overcome memory bandwidth limitations of traditional formats such as Compressed Sparse Row (CSR). Since consecutive values needed by Processing Elements (PEs) are not always stored contiguously within the same memory channel; thus, CSR causes ineffective bursts, channel conflicts, and wasted bandwidth, especially when accessing the last values of a row. As visualized in Figure 3, the C2SR format interleaves values in memory such that each Sparse Matrix A Loader (SpAL) can access all required values from a single dedicated memory channel, eliminating channel conflicts entirely. Furthermore, C2SR arranges data to ensure all fetched memory bursts are fully utilized – even at row boundaries – by placing the next needed values immediately after the current row ends. This design ensures that every burst fetched from memory is entirely effectual, maximizing memory bandwidth utilization and improving overall throughput. MatRaptor's high-level structure is composed of a set of PEs. Each PE is accompanied by an SpAL and Sparse Matrix B Loader (SpBL), which loads required values from memory and stage them for the PE through a series of FIFOs.

1) Room For Further Investigation: The prior work's analysis of MatRaptor's C2SR format has several critical limitations. First, the original evaluation measured bandwidth improvements only under idealized conditions, neglecting memory interference between different components accessing memory simultaneously that could reduce the practical benefits of C2SR. Second, prior work did not consider how using C2SR impacts the efficiency of writing outputs back to memory, which could constrain the accelerator's overall throughput as writing output back to memory becomes a bottleneck. Third, their experiments did not explore how the improvements in memory bandwidth utilization translate into runtime or performance improvements during accelerator operation. Lastly, the baseline CSR implementation was overly conservative, restricting memory accesses to 8-byte chunks to prevent crossing cache-line boundaries unnecessarily, potentially overstating the advantages attributed to C2SR. Consequently, the actual impact of irregular memory accesses, cross-component interference, and realistic operational scenarios on accelerator performance remains unclear.

2) Experimental Results: An important shortcoming in the prior evaluation was their uneven treatment of CSR and C2SR formats, which led to bandwidth comparisons under unrealistic conditions. To address this, we establish an experimental setup ensuring both CSR and C2SR formats have equal access to memory bandwidth, enabling a fair comparison where each can fully utilize available resources. Specifically, we implement two key changes. First, we configure Ramulator such that both CSR and C2SR have equal bandwidth access. For our base (C2SR) implementation, we split memory across multiple independent channels, assigning each PE to a dedicated channel. In the CSR implementation, we instead connect the PEs to all memory channels. This adds minimal overhead, since the base accelerator structure already includes a crossbar for the SpBLs to access all memory channels. Second, we ensure that all memory accesses in C2SR are entirely effectual, meaning each 32-byte burst from memory exclusively contains useful data. This adjustment contrasts with CSR, where memory accesses often include unused bytes due to alignment issues. By establishing these conditions, our analysis directly investigates how improvements in memory bandwidth utilization impact the actual performance of the accelerator, addressing previously overlooked interactions and providing a more realistic comparison between formats.

As shown in Figure 4, the experimental results indicate that CSR unexpectedly outperforms C2SR on runtime at

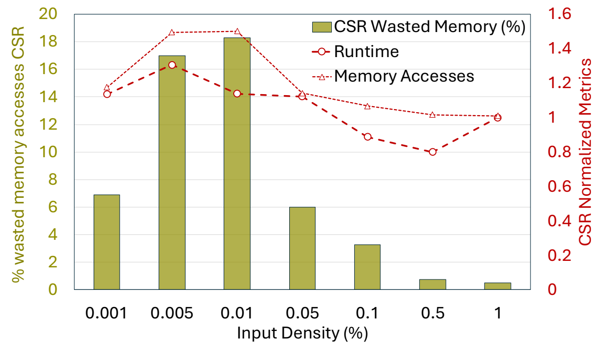


Fig. 4: The percentage of wasted memory accesses for CSR along with CSR's performance metrics normalized to the baseline (C2SR) over input density

mid-range sparsity levels (between 0.001 and 0.005). This observation is explained primarily by two interacting factors. First, at these sparsity levels, the incremental memory savings achieved by C2SR—reducing a few memory accesses between rows—become negligible relative to the total size of each row, diminishing its impact. This can be seen in the bars in Figure 4 showing the diminishing percentage of memory traffic that is wasted. Second, and more significantly, MatRaptor's implementation of C2SR constrains each PE to a single memory channel during writing. This restriction severely limits the write-back throughput, causing substantial PE idle cycles as the system waits for completed outputs to be written to memory before beginning new computations. The CSR implementation, by contrast, can exploit higher memory bandwidth due to its access to multiple channels simultaneously, reducing these memory-induced stalls.

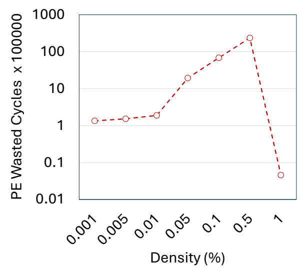


Fig. 5: C2SR: Number of PE cycles wasted waiting for memory writes over input density.

The increasing dependence that MatRaptor with C2SR has on write throughput can be seen in Figure 5, where we see a nearly exponential increase in the number of cycles wasted waiting for memory writes up until we reach 1% density. Beyond approximately 1% density, the accelerator's runtime transitions from memory-bound to computation-bound, as PE merge operations become the dominant factor limiting overall throughput rather than memory latency. This change is reflected in Figure 4, with the CSR and C2SR runtime becoming equal.

This interaction – between a memory layout optimization (C2SR) and architectural constraints imposed by the optimization (limited memory channels per PE) – reveals how optimizations can yield unintended bottlenecks when evaluated under realistic conditions. Our results clearly highlight the importance of thoroughly exploring interactions between memory optimizations and architectural components, as overlooking them can result in misleading conclusions about an optimization's true effectiveness. Future analyses of sparse accelerators should similarly prioritize understanding and explicitly modeling these interactions, ensuring conclusions reflect realistic operating scenarios.

B. Case Study 2: Dynamic Row Scheduler

As an **Orchestrate** optimization, Gamma's **Dynamic Row Scheduler** [5] addresses the challenges of parallelizing work between PEs and maximizing cache efficiency. Gamma's architecture consists of a set of asynchronous PEs connected to a shared cache – the FiberCache – which serves a dual role: (1) storing input matrix data and (2) temporarily holding partial outputs for quick reuse by the PEs

Gamma processes matrices in CSR format, computing results using a row-wise product approach. When input rows exceed the processing limit of a PE, they are tiled into smaller chunks and computed into partial sums. Partial sums are temporarily stored in the FiberCache until all tiles are processed, after which they are merged into a final output by a PE. The Dynamic Row Scheduler iterates over the rows of matrix A, scheduling computations dynamically to optimally balance load across PEs and minimizing the time that partial outputs stay in cache. By doing so, it improves cache efficiency, freeing space for input data and reducing memory accesses.

In contrast, a Static Scheduler would assign each PE to a fixed row before computation begins, ignoring tile constraints. This often results in partial outputs lingering in cache longer, limiting space for new input data and increasing the risk of cache spills. While the end result of the Dynamic Scheduler is a reduction in memory accesses, its primary function is reordering computation rather than directly optimizing memory access patterns – making it an Orchestrate optimization instead of an Access optimization.

1) Room For Further Investigation: Gamma provides some performance breakdowns for its components, but its evaluation of dynamic scheduling is limited. Prior work compares the fully dynamic scheduler (where all jobs are mapped to all PEs) to a variant that maps each output row to a single PE. While this is a reasonable comparison, it does not directly test whether dynamic scheduling itself improves performance. A more informative experiment would compare the dynamic scheduler against a static scheduler that pre-assigns rows to PEs before computation begins.

Additionally, Gamma's evaluation is limited to a single sparse matrix and only reports memory traffic composition between the cache and DRAM, concluding that the dynamic scheduler reduces the need to save and load partial matrices from memory. To fully assess this optimization's impact, we extend the evaluation to multiple input matrices and analyze additional performance metrics, including runtime, cache composition, and memory accesses.

2) Experimental Results: Figure 6 shows that the Dynamic Row Scheduler improves performance as matrices become denser, an unexpected outcome for an optimization designed for sparse workloads. This effect arises because the scheduler

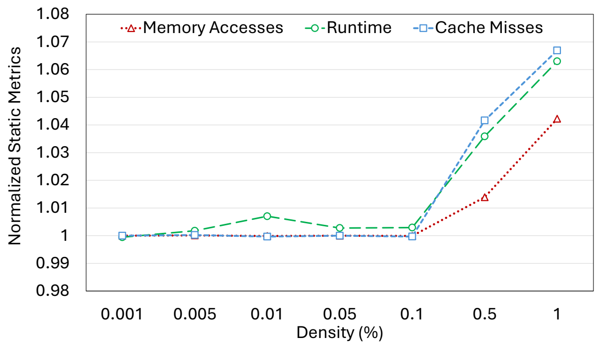


Fig. 6: The static scheduler normalized to the baseline (dynamic) in terms of memory accesses, runtime, and cache misses, over input density.

reduces the number of partial matrices stored in the cache, a benefit that becomes more critical as density increases.

In extremely sparse matrices, each PE can often compute an entire output row in a single processing round, eliminating the need to store partial results. However, as matrix density increases, partial sums accumulate and evict rows of the input matrices, making cache management essential for sustaining performance. The Dynamic Scheduler mitigates this issue by parallelizing tiled row computation, ensuring partial outputs remain in cache only as long as necessary before being used and evicted.

By contrast, a static scheduler in its place inefficiently assigns each PE to a single row, often causing partial outputs to persist in cache longer than necessary, limiting available space for input data, and increasing cache pressure. This scheduling inefficiency leads to more frequent memory stalls and reduced performance.

The effectiveness of this optimization depends on the structure of the input matrix, particularly the row nonzero count. Matrices with many nonzeros concentrated in a few rows tend to generate more partial outputs, leading to greater cache pressure. This effect is particularly pronounced in non-square sparse matrices with long rows or matrices with a highly uneven distribution of nonzeros.

Figure 7 further supports this conclusion by comparing cache composition across all SuiteSparse inputs. On average, the dynamic scheduler only slightly reduces the fraction of cache occupied by partial outputs. However, when analyzing only matrices with at least one row exceeding 64 nonzeros—where partial sums become significant—the effect is much stronger: in the unoptimized case, nearly 10% of the cache is occupied by partial outputs, whereas the dynamic scheduler cuts this amount in half, to 5%. While this may seem like a small difference, in large workloads, this translates to thousands (or even hundreds of thousands) of output rows avoiding unnecessary evictions, since partial outputs generally have a low lifespan inside the FiberCache.

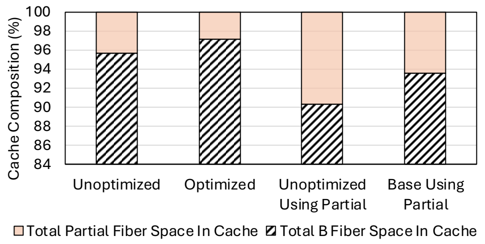


Fig. 7: Gamma Cache Composition with and without Scheduler: Comparison over All SuiteSparse Matrices [42] and Matrices Using Partial Outputs ("Using Partial")

These findings reinforce the value of the dynamic scheduler while highlighting an important takeaway: optimization effectiveness is often tied to specific workload characteristics. While certain optimizations impact compute at all stages, this case study shows an optimization that only comes into play when partial sums are required in computation. This distinction is crucial for researchers applying similar optimizations, as it underscores the need to analyze workload-specific constraints, which could lead to optimizations being underutilized.

C. Case Study 3: Array-Based Merger

The **Array-Based Merger**, introduced in SpArch [6], is a **Merge** optimization that improves throughput by merging multiple values in a cycle instead of a single value. SpArch, which relies on the outer product method, heavily depends on merging partial matrices when computing output. These partial matrices are represented as streams of (value, row, column) tuples in memory, and in order to ensure that inputs are merged in the correct order, they are sorted by their row and then column index. Normally, merging these partial matrices would proceed one value at a time, with the merge component comparing the first value(s) of each input stream to send one or the other, or adding them if they have the same coordinates. The Array-Based Merger instead uses a comparator array – a grid of comparator circuits – to merge multiple values in a single cycle, directly improving the throughput of this operation. For example, a 16 × 16 Array-Based Merger would look at the first 16 values from both input streams, and output 16 merged values in a single cycle. Within SpArch, these mergers are placed into a merge tree, with one merger operating at each level to merge values and place them into the FIFOs of the layers above them.

SpArch's dataflow starts with the Matrix A Fetcher, which loads columns of matrix A from memory in a condensed CSR format. These fetched elements enter the Lookahead FIFO, which temporarily stores upcoming elements from the A matrix to predict future access patterns. Using column indices stored in this Lookahead FIFO, the B Matrix Prefetcher anticipates and preloads necessary rows from matrix B into an on-chip buffer, employing a near-optimal replacement strategy based on this information. Critically to our results, fetching and multiplication are not done in parallel with merging. This means that the accelerator is either fetching and multiplying values to stage them for the merge tree, or merging those values, never doing both simultaneously.

Fetched data from the A matrix and B matrix stream into a multiplier array, which performs parallel multiplications to generate partial matrices. These partial matrices flow

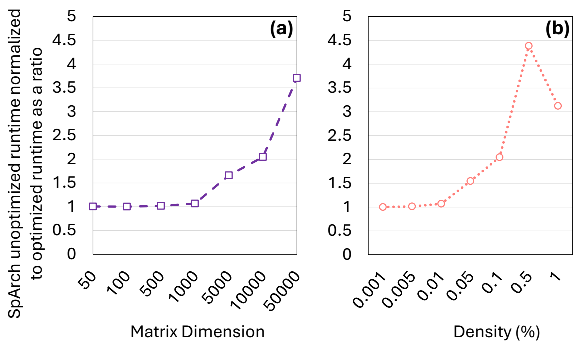


Fig. 8: SpArch's Array-Based Merger on the runtime when (a) size and (b) sparsity vary.

directly into the Merge Tree, a specialized hardware structure composed of Array-Based Mergers and FIFOs arranged hierarchically, merging multiple sorted partial matrices efficiently.

1) Room for Further Investigation: SpArch does take steps to report the performance of the Array-Based Merger, but there are still a few pieces missing. The work reported the *GFLOPS* and number of DRAM accesses achieved by varying the comparator array's size. This means we get a good idea of the average runtime improvement of the accelerator between 1×1 and 16×16 multipliers (since the amount of computation is the same), but these values are a combined result of the accelerator evaluated across over 20 datasets specified in another work [26]. Insights into the optimization's overall effectiveness across varying levels of sparsity and matrix sizes, as well as potential bottlenecks within the merge tree caused by differences in input types, remain largely unexplored.

2) Experimental Results: Our analysis of SpArch's Array-Based Merger evaluates the efficiency of merge operations and how it interacts with varying input densities. Figure 8 shows how the optimization affects runtime across matrices of different sizes and sparsities. Generally, denser and larger matrices involve more merge operations, offering greater opportunities for efficiency improvements using the optimized merger.

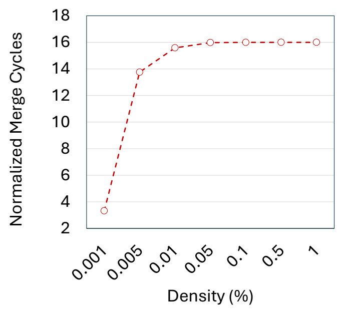


Fig. 9: Ratio of number of cycles spent merging between unoptimzed merger and Array-Based Merger over input density.

This relationship becomes clearer when we look at the normalized number of merge cycles in Figure 9. As input density increases (or sparsity decreases), the optimized merger approaches – but does not quite reach – a 16× improvement in merge cycles compared to the unoptimized version. This limit makes intuitive sense because the Array-Based Merger merges 16 values at once, naturally setting an upper bound on improvement. At higher densities, the merger can achieve significant efficiency gains, despite some overhead from stalls and the merge tree structure itself. Notably, the runtime improvements do not scale with the efficiency of the optimization over the baseline, drawing a far more gradual curve of performance improvement. Additionally, at 1% density, there is a noticeable mismatch between the two graphs – runtime improvements slow down even though the improvements to the number of merge cycles remain consistent. Both of these indicate that the performance improvement or degradation that we see as we vary density and size are not a function of the optimization's effectiveness.

As density increases, the improvements brought by the optimized merger to total runtime initially grows but subsequently decreases, as shown in Figure 8. Despite the merger's capability to consistently achieve near-maximum efficiency, its visible benefits become constrained by other runtime factors, such as data movement or other on-chip computations. Because SpArch does not perform merging in parallel with other operations, merging competes directly with memory fetching and multiplication tasks for runtime. In low-density situations, these tasks occupy significant runtime in comparison with merging, which remains a simple operation. Gradually, as density and size increase, merging becomes more complex anjd takes up a larger amount of the total runtime of the accelerator. After 1% density, partial outputs being saved to and loaded from DRAM take up a large portion of runtime, reducing the amount of runtime spent merging and limiting the performance improvements that the Array-Based Merger brings. Simply, the more the accelerator performs tasks outside of merging, the smaller the overall impact of the merge optimization on runtime, causing the performance metrics to converge. Though this pattern of the optimization's effectiveness being reliant on input may seem similar to the Dynamic Row Scheduler introduced in the previous case study, they are not the same. The Array-Based Merger does not perform better or worse on the input. The performance trends we see, paradoxically, are dictated solely by the performance of the accelerator on all computations excluding merging, since the portion of runtime allocated to merging is determined by them.

Our findings show that relying solely on overall runtime metrics can conceal important performance details and lead to misleading conclusions about an optimization's effectiveness. By examining how merging efficiency changes with varying input densities, we pinpoint critical moments when other parts of the system limit direct benefits from optimization. Thus, accurately evaluating hardware optimizations requires looking at both specific performance metrics and how these metrics came to be in the context of the overall system, allowing for better decisions when designing or choosing optimizations for practical use.

VI. Analysis of Optimization Categories

In this section, we go over the optimization categories, and their effectiveness as a whole, to see if we can find broad patterns in how each category interacts with varying inputs. We first evaluate each optimization category with respect to input sparsity and size, investigate a pattern and tradeoff we see across our evaluations, and then dive into their performance on the real-world sparse matrices.

A. Sparsity & Size Evaluations

One of the primary purposes of SpGEMM accelerators is to scale well with large and sparse matrices. Our optimizations aim to further this goal, but due to the differences in scope between each optimization category, we expect each category to respond differently to varying inputs. Figure 10 summarizes our findings at a high level, presenting the median runtime and memory access improvements across the three categories for synthetic matrices with varying sparsity and size.

As shown in Figure 10 (a) and (c), Merge optimizations perform best on large and dense inputs, with Access optimizations being more effective on smaller and sparser matrices and Orchestrate optimizations shadowing Merge optimizations to a lesser extent. An important secondary observation is the sharp increase in performance improvements by Merge optimizations, rather than a smooth linear rise like we see in Case Study 3. If the number of operations alone determined this behavior, we would expect a gradual increase. Instead, the sharp jump followed by stabilization reflects the role of operational intensity – the number of operations per byte fetched from memory. This relationship will be further explored in the following section

Access optimizations behave mostly as expected, outperforming other categories in terms of memory access reduction, as shown in Figure 10 (b) and (d). The "hump" pattern stems from a lack of opportunity to optimize the smaller and sparser inputs, followed by solid improvements on middling inputs, and then a degradation of performance as the primary form of memory movement becomes partial outputs to and from memory, which are far larger and do not have as much room to be optimized by Access optimizations since they are single-use and unique.

Finally, as seen in Figure 10, the impact of Orchestrate optimizations increases with input complexity, shadowing Merge optimizations for larger, denser matrices, while maintaining some improvements across smaller, sparser inputs. This category enhances operational efficiency by improving data locality and synchronization between components, which can both reduce idle times and improve throughput. Unlike Access or Merge optimizations, which are more directly tied to memory bandwidth or peak performance limits, Orchestrate optimizations adjust how inputs are distributed and processed within the accelerator, allowing the system to adapt more effectively to all inputs. That is, Orchestrate optimizations improve performance regardless of whether the accelerator is compute or memory bound, and can impact either one depending on the optimization.

B. Benefits and Tradeoffs of Optimization Categories

A critical insight from our results, supported by our case studies, is the significant but context-dependent performance impact of Merge optimizations across accelerators, as shown in Figure 10. Merge optimizations, such as SpArch's Array-Based

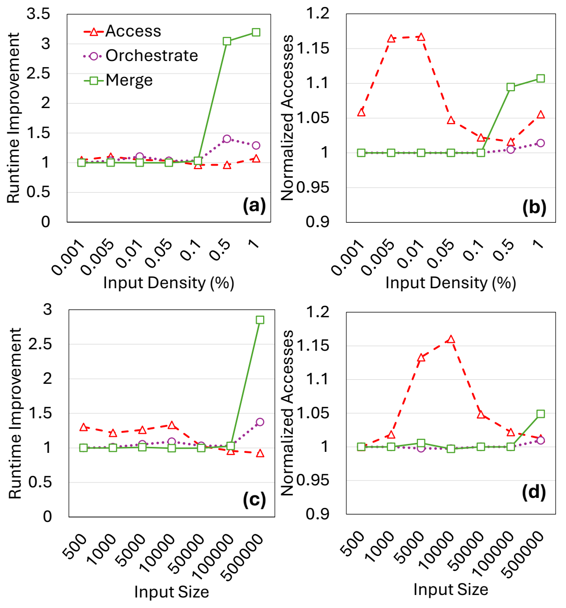


Fig. 10: Median Access, Orchestrate, and Merge unoptimized runtime and memory accesses normalized to baseline when density and size vary.

Merger, often provide straightforward, large performance improvements by enhancing merging efficiency and reducing computation cycles per operation. These optimizations directly increase achievable GFLOPS, leading to notable runtime gains for larger and denser inputs, particularly in compute-bound scenarios where internal computation dominates runtime. However, despite their theoretical impact, the improvements to Merge optimizations can fail to translate fully into overall accelerator performance improvements. This limitation arises because accelerators frequently remain memory-bound, rather than compute-bound in computation, with on-chip computation taking up a smaller portion of runtime compared to the challenges of memory movement. Consequently, while Merge optimizations significantly improve on-chip computations, their broader impact is limited unless compute dominates runtime, as observed in Case Study 3.

Access optimizations exhibit a complementary but distinct performance profile, generally offering moderate and consistent improvements by reducing off-chip memory traffic. These optimizations prove effective across a wide range of sparse inputs common to these accelerators. Additionally, they tend to greatly improve memory reuse and memory bandwidth utilization, even if they do not have an outstanding impact on runtime. Mirroring Merge optimizations, their impact on runtime diminishes sharply when accelerators become heavily compute-bound, as improvements to memory efficiency no longer meaningfully affect overall performance. A particular limitation of Access optimizations arises in scenarios involving partial outputs, which often dominate memory traffic. These partial outputs, characterized by limited reuse and challenging on-chip storage requirements, prove difficult for Access optimizations to effectively address, thereby constraining their usefulness in such contexts.

Orchestrate optimizations tend to function primarily as facilitators, providing modest yet stable improvements that enhance overall accelerator efficiency rather than significantly altering computational or memory performance. As exemplified by the Dynamic Row Scheduler, these optimizations focus on compensating in scenarios where Merge and Access optimizations fall short, such as managing partial output movement efficiently and efficiently parallelizing work. Though Orchestrate optimizations exhibit robustness across both memory- and compute-bound regimes, their impact remains comparatively limited. They do not define accelerator performance; instead, they serve as supportive optimizations that complement the strengths and mitigate the weaknesses of Merge and Access optimizations.

Our results emphasize the necessity of balanced integration across optimization categories: Merge optimizations targeting peak computational efficiency; Access optimizations reducing memory bandwidth demands; and Orchestrate optimizations dynamically allocating internal resources to sustain performance across diverse workloads. Each optimization category inherently carries specific trade-offs: Merge optimizations risk diminishing returns as computational limits are approached; Access optimizations become irrelevant when accelerators are strongly compute-bound or dominated by partial outputs; and Orchestrate optimizations consistently deliver lower-impact improvements, facilitating rather than defining performance.

Formally, we redefine these optimization categories accordingly: Merge optimizations as enhancements targeting internal computational efficiency in compute-bound scenarios; Access optimizations as methods improving memory efficiency primarily in memory-bound contexts; and Orchestrate optimizations as dynamic facilitators, optimizing resource allocation and data movement across varying operational intensities.

Future work should explore adaptive optimization strategies that dynamically combine these categories based on real-time workload profiling. Developing predictive techniques to anticipate bottleneck transitions could allow proactive optimization, mitigating the limitations inherent in any single optimization type. Investigating novel methods explicitly addressing the interaction between computational throughput and memory efficiency could further enhance accelerator performance, particularly in overcoming observed performance plateaus. Ultimately, accelerators capable of dynamically shifting optimization strategies based on real-time workload characteristics can effectively maximize performance across diverse, realistic scenarios.

C. Evaluation on SuiteSparse datasets

Figures 11 (a) and (b) show which of the optimization categories has the best median performance over each of the SuiteSparse matrices, categorized by size and type, respectively. In this plot, the matrices that these accelerators are most likely to encounter tend to the bottom right of the scatterplot.

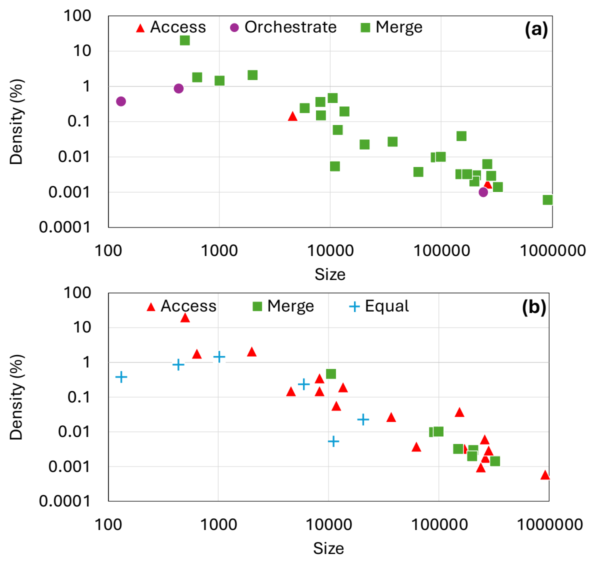


Fig. 11: SuiteSparse matrices, plotted by their density and size, and colored based on which optimization category has the (a) highest median runtime improvement and (b) the highest median memory access improvement

These graphs mostly confirm our expectations, with Merge optimizations dominating runtime across the board, and Access optimizations holding the advantage in memory accesses. The lack of observable patterns in the memory access findings indicate that matrix shape likely skews the results, but an analysis of how matrix shape interacts with Access optimizations is deeply optimization-specific and is outside the scope of this paper.

Despite this, these findings reaffirm our previous findings about the optimization categories. Keeping in mind that our SuiteSparse matrices tend towards being small and dense or large and sparse, Merge optimizations are clearly critical for these types of matrices that strain compute resources, while Access optimizations lean more towards improving resource efficiency. Orchestrate optimizations are rarely the best performing optimization category, but tend towards consistency across workloads with our case study optimization (Section V-B) being an exception rather than the norm. Together, these optimizations form a robust framework for scaling SpGEMM accelerators across diverse applications, from synthetic benchmarks to complex real-world data. Future work could expand on these findings by exploring interactions between matrix shape, dataflow orchestration, sensitivity to operational intensity and optimization efficacy.

VII. Conclusions

SpGEMM accelerators have potential to play a pivotal role in various application domains, yet their evaluation often overlooks the nuanced contributions of individual optimizations. This paper introduces a self-comparison methodology, enabling the isolation and analysis of these contributions. Through our experiments, we have demonstrated how different optimization categories – Access, Merge, and Orchestrate – perform under varying input sparsity and size, as well as in real-world conditions. Our results highlight critical trends: the dominance of Access optimizations for sparse or small inputs, the rising significance of Merge optimizations as inputs become denser, and the unique adaptability of Orchestrate optimizations.

These findings underline the importance of a holistic evaluation approach that accounts for the interplay between optimizations and input characteristics. Notably, our work reveals how certain optimizations can show immense value when their strengths are played to, as well as have little value at their weakest, reinforcing the necessity of comprehensive evaluation frameworks.

By dissecting and evaluating optimizations within sparse accelerators, this paper investigates the strengths and weaknesses of these optimizations and categories, offering a set of useful insights for future researchers. Our research not only deepens understanding of SpGEMM accelerator performance but also establishes a framework for developing adaptive and efficient accelerators. As SpGEMM accelerators increasingly address critical computational workloads, the findings presented here provide a foundation for further innovation and more targeted optimization strategies tailored to the ever-expanding diversity of real-world data.

Beyond static designs, the findings from La Superba extend to dynamically reconfigurable systems (DRS), which face added complexity from adapting to diverse workloads in real-time. DRS demands modular and flexible optimization strategies capable of responding to varying input characteristics and computational requirements. By providing detailed insights into how individual optimizations and categories perform across a wide spectrum of conditions, La Superba lays the groundwork for creating hardware systems that dynamically reconfigure at an optimization level. This opens up exciting opportunities for future research, leveraging the modularity and adaptability of sparse accelerators to address the challenges posed by dynamic workloads.

In conclusion, as SpGEMM accelerators tackle critical computational tasks, this paper's methodologies provide a foundation for innovation in static and reconfigurable designs, advancing efficient, workload-aware hardware tailored to real-world needs.

Acknowledgments

This paper is a part of the work supported by the U.S. Department of Energy (DoE), Office of Science, Office of Advanced Scientific Computing Research (ACSR) the Early Career Research Program (ECRP), under Award Number DE-SC0024079.

References

[1] N. Bell, S. Dalton, and L. N. Olson, "Exposing fine-grained parallelism in algebraic multigrid methods," *SIAM Journal on Scientific Computing*, vol. 34, no. 4, pp. C123–C152, 2012. [Online]. Available: https://doi.org/10.1137/110838844

[2] G. Ballard, C. M. Siefert, and J. J. Hu, "Reducing communication costs for sparse matrix multiplication within algebraic multigrid," *SIAM J. Sci. Comput.*, vol. 38, 2015. [Online]. Available: https://api.semanticscholar.org/CorpusID:18274179

[3] J. S. Mueller-Roemer, C. Altenhofen, and A. Stork, "Ternary sparse matrix representation for volumetric mesh subdivision and processing on gpus," *Comput. Graph. Forum*, vol. 36, no. 5, p. 59–69, Aug. 2017. [Online]. Available: https://doi.org/10.1111/cgf.13245

[4] N. Bock, M. Challacombe, and L. V. Kalé, "Solvers for $\mathcal{O}(n)$ electronic structure in the strong scaling limit," 2015. [Online]. Available: https://arxiv.org/abs/1403.7458

[5] G. Zhang, N. Attaluri, J. S. Emer, and D. 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*, 2021, pp. 687–701.

[6] Z. Zhang, H. Wang, S. Han, and W. J. Dally, "Sparch: Efficient architecture for sparse matrix multiplication," in *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA).* IEEE, 2020, pp. 261–274.

[7] N. Srivastava, H. Jin, J. Liu, D. Albonesi, and Z. Zhang, "Matraptor: A sparse-sparse matrix multiplication accelerator based on row-wise product," in *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO).* IEEE, 2020, pp. 766–780.

[8] X. He, S. Pal, A. Amarnath, S. Feng, D.-H. Park, A. Rovinski, H. Ye, Y. Chen, R. Dreslinski, and T. Mudge, "Sparse-tpu: Adapting systolic arrays for sparse matrices," in *Proceedings of the 34th ACM international conference on supercomputing*, 2020, pp. 1–12.

[9] X. Zhou, Z. Du, Q. Guo, S. Liu, C. Liu, C. Wang, X. Zhou, L. Li, T. Chen, and Y. 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).* IEEE, 2018, pp. 15–28.

[10] B. Wang, S. Ma, S. Luo, L. Wu, J. Zhang, C. Zhang, and T. Li, "Spargd: A sparse gemm accelerator with dynamic dataflow," *ACM Transactions on Design Automation of Electronic Systems*, vol. 29, no. 2, pp. 1–32, 2024.

[11] Y. Tortorella, L. Bertaccini, L. Benini, D. Rossi, and F. Conti, "Redmule: A mixed-precision matrix–matrix operation engine for flexible and energy-efficient on-chip linear algebra and tinyml training acceleration," *Future Generation Computer Systems*, vol. 149, pp. 122–135, 2023.

[12] V. Sze, Y.-H. Chen, T.-J. Yang, and J. S. Emer, *Efficient processing of deep neural networks.* Springer, 2020.

[13] S. Zeng, Y. Lin, S. Liang, J. Kang, D. Xie, Y. Shan, S. Han, Y. Wang, and H. Yang, "A fine-grained sparse accelerator for multi-precision dnn," in *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 185. [Online]. Available: https://doi.org/10.1145/3289602.3293964

[14] G. Li, W. Xu, Z. Song, N. Jing, J. Cheng, and X. Liang, "Ristretto: An atomized processing architecture for sparsity-condensed stream flow in cnn," in *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2022, pp. 1434–1450.

[15] A. Azad, A. Buluç, and J. Gilbert, "Parallel triangle counting and enumeration using matrix algebra," in *2015 IEEE International Parallel and Distributed Processing Symposium Workshop.* IEEE, 2015, pp. 804–811.

[16] J. R. Gilbert, S. Reinhardt, and V. B. Shah, "High-performance graph algorithms from parallel sparse matrices," in *Proceedings of the 8th International Conference on Applied Parallel Computing: State of the Art in Scientific Computing*, ser. PARA'06. Berlin, Heidelberg: Springer-Verlag, 2006, p. 260–269.

[17] A. Buluç and J. R. Gilbert, "The combinatorial blas: design, implementation, and applications," *Int. J. High Perform. Comput. Appl.*, vol. 25, no. 4, p. 496–509, Nov. 2011. [Online]. Available: https://doi.org/10.1177/1094342011403516

[18] S. R. Agrawal, C. M. Dee, and A. R. Lebeck, "Exploiting accelerators for efficient high dimensional similarity search," in *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '16. New York, NY, USA: Association for Computing Machinery, 2016. [Online]. Available: https://doi.org/10.1145/2851141.2851144

[19] J. Kim, M. Jang, H. Nam, and S. Kim, "Harp: Hardware-based pseudo-tiling for sparse matrix multiplication accelerator," in *Proceedings of the*

56th Annual IEEE/ACM International Symposium on Microarchitecture, 2023, pp. 1148–1162.

[20] G. E. Moon, H. Kwon, G. Jeong, P. Chatarasi, S. Rajamanickam, and T. Krishna, “Evaluating spatial accelerator architectures with tiled matrix-matrix multiplication,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 33, no. 4, pp. 1002–1014, 2021.

[21] J. Gao, W. Ji, F. Chang, S. Han, B. Wei, Z. Liu, and Y. Wang, “A systematic survey of general sparse matrix-matrix multiplication,” *ACM Computing Surveys*, vol. 55, no. 12, pp. 1–36, 2023.

[22] J. Mellor-Crummey, D. Whalley, and K. Kennedy, “Improving memory hierarchy performance for irregular applications,” in *Proceedings of the 13th International Conference on Supercomputing*, ser. ICS ’99. New York, NY, USA: Association for Computing Machinery, 1999, p. 425–433. [Online]. Available: https://doi.org/10.1145/305138.305228

[23] B. Asgari, R. Hadidi, J. Dierberger, C. Steinichen, A. Marfatia, and H. Kim, “Copernicus: Characterizing the performance implications of compression formats used in sparse workloads,” in *2021 IEEE International Symposium on Workload Characterization (IISWC).* IEEE, 2021, pp. 1–12.

[24] E. Qin, A. Samajdar, H. Kwon, V. Nadella, S. Srinivasan, D. Das, B. Kaul, and T. 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).* IEEE, 2020, pp. 58–70.

[25] F. Muñoz-Martínez, R. Garg, M. Pellauer, J. L. Abellán, M. E. Acacio, and T. 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*, 2023, pp. 252–265.

[26] S. Pal, J. Beaumont, D.-H. Park, A. Amarnath, S. Feng, C. Chakrabarti, H.-S. Kim, D. Blaauw, T. Mudge, and R. Dreslinski, “Outerspace: An outer product based sparse matrix multiplication accelerator,” in *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA).* IEEE, 2018, pp. 724–736.

[27] Z. Li, J. Li, T. Chen, D. Niu, H. Zheng, Y. Xie, and M. 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*, 2023, pp. 747–761.

[28] C. K. Vadlamudi and B. Asgari, “Electra: Eliminating the ineffectual computations on bitmap compressed matrices,” *IEEE Computer Architecture Letters*, 2024.

[29] A. Gerami and B. Asgari, “Gust: Graph edge-coloring utilization for accelerating sparse matrix vector multiplication,” in *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2024.

[30] D. Ramchandani, B. Asgari, and H. Kim, “Spica: Exploring fpga optimizations to enable an efficient spmv implementation for computations at edge,” in *2023 IEEE International Conference on Edge Computing and Communications (EDGE).* IEEE, 2023, pp. 36–42.

[31] K. Hegde, H. Asghari-Moghaddam, M. Pellauer, N. Crago, A. Jaleel, E. Solomonik, J. Emer, and C. W. Fletcher, “Extensor: An accelerator

[37] H. Huang and E. Chow, “Exploring the design space of distributed parallel sparse matrix-multiple vector multiplication,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 35, no. 11, pp. 1977–1988, 2024.

for sparse tensor algebra,” in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2019, pp. 319–333.

[32] S. Dave, R. Baghdadi, T. Nowatzki, S. Avancha, A. Shrivastava, and B. Li, “Hardware acceleration of sparse and irregular tensor computations of ml models: A survey and insights,” *Proceedings of the IEEE*, vol. 109, no. 10, pp. 1706–1752, 2021.

[33] S. Han, X. Liu, H. Mao, J. Pu, A. Pedram, M. A. Horowitz, and W. J. Dally, “Eie: Efficient inference engine on compressed deep neural network,” 2016. [Online]. Available: https://arxiv.org/abs/1602.01528

[34] J. O. Tørring, J. C. Meyer, and A. C. Elster, “Autotuning benchmarking techniques: A roofline model case study,” 2021. [Online]. Available: https://arxiv.org/abs/2103.08716

[35] Y. N. Wu, P.-A. Tsai, A. Parashar, V. Sze, and J. S. Emer, “Sparseloop: An analytical approach to sparse tensor accelerator modeling,” 2023. [Online]. Available: https://arxiv.org/abs/2205.05826

[36] C. Y. Lin, Z. Zhang, N. Wong, and H. K.-H. So, “Design space exploration for sparse matrix-matrix multiplication on fpgas,” in *2010 International Conference on Field-Programmable Technology*, 2010, pp. 369–372.

[38] U. Bakhtiar, H. Hosseini, and B. Asgari, “Acamar: A dynamically reconfigurable scientific computing accelerator for robust convergence and minimal resource underutilization,” in *2024 57th IEEE/ACM International Symposium on Microarchitecture (MICRO).* IEEE, 2024, pp. 1601–1616.

[39] V. STRASSEN, “Gaussian elimination is not optimal.” *Numerische Mathematik*, vol. 13, pp. 354–356, 1969. [Online]. Available: http://eudml.org/doc/131927

[40] S. Milaković, O. Selvitopi, I. Nisa, Z. Budimlić, and A. Buluc, “Parallel algorithms for masked sparse matrix-matrix products,” in *Proceedings of the 51st International Conference on Parallel Processing*, ser. ICPP ’22. New York, NY, USA: Association for Computing Machinery, 2023. [Online]. Available: https://doi.org/10.1145/3545008.3545048

[41] F. G. Gustavson, “Two fast algorithms for sparse matrices: Multiplication and permuted transposition,” *ACM Trans. Math. Softw.*, vol. 4, no. 3, p. 250–269, sep 1978. [Online]. Available: https://doi.org/10.1145/355791.355796

[42] T. A. Davis and Y. Hu, “The university of florida sparse matrix collection,” *ACM Trans. Math. Softw.*, vol. 38, no. 1, dec 2011. [Online]. Available: https://doi.org/10.1145/2049662.2049663

[43] H. Luo, Y. C. Tuğrul, F. N. Bostancı, A. Olgun, A. G. Yağlıkçı, , and O. Mutlu, “Ramulator 2.0: A Modern, Modular, and Extensible DRAM Simulator,” 2023.

[44] K. Asifuzzaman, M. Abuelala, M. Hassan, and F. J. Cazorla, “Demystifying the characteristics of high bandwidth memory for real-time systems,” in *2021 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*, 2021, pp. 1–9.

[45] K. Macdonald, C. Nitta, M. Farrens, and V. Akella, “Pdg_gen: A methodology for fast and accurate simulation of on-chip networks,” *IEEE Transactions on Computers*, vol. 63, no. 3, pp. 650–663, 2012.

[46] CoffeeBeforeArch, “Writing a trace-based cache simulator,” 2020. [Online]. Available: https://coffeebeforearch.github.io/2020/12/16/cache-simulator.html

[47] Y.-C. Chen, T. Seidl, N. Hölscher, C. Hakert, M. D. Truong, J.-J. Chen, J. P. C. de Lima, A. A. Khan, J. Castrillon, A. Nezhadi ***et al.***, “Modeling and simulating emerging memory technologies: A tutorial,” *arXiv preprint arXiv:2502.10167*, 2025.