Procyon: Promoting Fine-Grain Multi-Tenancy to Optimize Sparse Streaming Accelerators

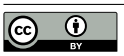
Ubaid Bakhtiar*, Jeremy Sha, Helya Hosseini, and Bahar Asgari
University of Maryland, College Park

Abstract

Sparsity has become a defining characteristic of modern workloads, motivating the development of specialized accelerators to mitigate the challenges inherent to sparsity. Sparse streaming accelerators have proved to be an effective solution, yet current designs are restricted to single-workload execution. Additionally, they exhibit significant underutilization in processing elements (PEs) due to their non-zero scheduling strategies. To address these limitations, we propose Procyon, a fine-grain multi-tenancy framework that fuses the PE instruction streams of multiple workloads into a unified execution schedule. This allows instructions from different workloads to execute on the same PE, thereby, improving its utilization and enabling concurrent execution of multiple workloads on a single accelerator instance. We evaluate Procyon on AMD Alveo U55C FPGA using the SuiteSparse workloads and show that it substantially reduces PE underutilization that results in 3× speedup over state-of-the-art sparse streaming accelerators (Serpens and Chasoň), and reaches a peak throughput of 61.2 GFLOP/s.

1 Introduction

Sparsity has emerged as a critical feature in modern applications, spanning scientific computing [1, 8, 9, 12, 15, 28], graph analytics [6, 19, 20], bioinformatics [4, 23] and natural language processing [7, 17, 18, 21, 29, 33]. The existence of sparsity in data leads to inherent randomness, resulting in irregular memory access, inefficient data reuse and suboptimal resource utilization, particularly in traditional architectures such as CPUs and GPUs. In response to these challenges, sparse streaming accelerators [3, 13, 22, 24, 25, 31, 32] have gained substantial attention in accelerating sparse kernels, such as sparse matrix-vector multiplication (SpMV) and sparse matrix-dense matrix multiplication (SpMM). These accelerators offer promising solutions for optimizing sparse kernels. They have demonstrated substantial performance gains over CPUs and GPUs, positioning them as increasingly mainstream solutions for sparse computations. The key factor behind the enhanced performance of streaming accelerators is their dataflow execution model. In this model, there is a direct correspondence between how data is scheduled in the memory and how it is mapped onto the processing elements (PEs), allowing for efficient and tailored processing of sparse data. This execution model eliminates the need for explicit instruction storage and fetching, streamlining the data execution pipeline by removing intermediate buffering and idle stages.



DAC '26, Long Beach, CA, USA

ACM ISBN 979-8-4007-2254-7/2026/07
https://doi.org/10.1145/3770743.3804076

For example, Serpens [24] is a recent sparse streaming accelerator that introduces a cyclic, row-wise scheduling scheme known as PE-aware non-zero scheduling. This technique efficiently maps the non-zero elements of sparse inputs to off-chip memory channels. Building upon this, the state-of-the-art accelerator, Chasoň [3], introduces a cross-HBM channel out-of-order (OoO) scheduling mechanism, or CrHCS, to further enhance PE-aware non-zero scheduling by allowing data migration across High Bandwidth Memory (HBM) channels. This improvement ensures a more compact and efficient placement of non-zero elements across off-chip memory channels. Despite the significant performance gains offered by sparse streaming accelerators, opportunities for improvements still remain. We have identified two critical challenges that need to be addressed to enhance their efficiency and usability in practical scenarios:

Challenge 1. State-of-the-art sparse streaming accelerators are currently designed to handle a single workload per execution instance, meaning they can process only one workload at a time. This limitation significantly hampers their efficiency in multi-user environments, such as data centers, where multiple users require simultaneous access to sparse accelerators.

Challenge 2. These accelerators fail to fully exploit the available computational resources, such as processing elements (PEs), because of inefficient data scheduling strategies. This results in considerable number of stalls in the execution pipeline and ultimately, PE underutilization.

* To address these challenges, we propose Procyon†, a fine-grain multi-tenancy approach that unifies the instructions of multiple workloads in to one execution schedule, allowing them to be processed by a single PE. Procyon achieves this by replacing the stalls in the execution pipeline with instructions associated with another workload. By consolidating the instructions from different workloads/tasks onto a single PE, our approach removes the need for multiple instances of sparse streaming accelerators as well as reduce the PE underutilization, enabling the efficient handling of multiple concurrent requests. We evaluate Procyon on state-of-the-art sparse streaming accelerators, Serpens [24] and Chasoň [3] for SpMV kernel based on their AMD Alveo U55C implementation. Our evaluation on SuiteSparse [5] workloads shows that Procyon substantially reduces PE underutilization, achieves up to 3× speedup over Serpens [24] and Chasoň [3], and reaches a peak throughput of 61.2 GFLOP/s.

2 Background and Challenges

2.1 Sparse Accelerators

General purpose CPUs and GPUs fail to fully exploit the benefits of sparsity as a result of irregular memory accesses and poor data

*Corresponding author: ubaidb@umd.edu
†Procyon [/prəʊsɪən/] is the eighth-brightest star in the night sky and the brightest star in the constellation Canis Minor. In ancient greek, Procyon also means 'raccoon'.

reuse. To deal with that, sparse accelerators have become a popular solution for sparse workloads. Several of these accelerators focus on data partitioning and distribution, for example, Quartz [10] is a recent reconfigurable, distributed-memory accelerator for sparse applications, which enables fine-grain data partitioning across the tiles of a single workload. Segin [16] is another example that enables multi-threading across the chunks of matrices to efficiently utilize systolic arrays. Another category of accelerators targets improving on-chip data placement, such as Pipirima [2], which uses predictions to distribute sparse data evenly on the on-chip memory blocks and optimize SpMV and SpMM. Some other sparse accelerators co-design the architecture based on their novel compression formats such as MatRaptor [26], Tensaurus [27] and SparTen [11]. More recent techniques, such as Misam [32] and Bootes [31], leverage machine learning to enhance data reuse and reduce memory bottlenecks. Sparse streaming accelerators are another class of accelerators where the instructions are generated based on the way data is scheduled in the memory. For example, Serpens [24] and Chasoň [3] are the two recent FPGA-based sparse streaming accelerators that schedule the non-zero values in HBM channels to ensure parallel instruction execution without any intermediate buffering. Sparse streaming accelerators have shown considerable performance improvements over many existing sparse acceleration techniques, making them a practical and promising approach for handling sparse workloads.

2.2 Single Workload Execution Model

A key limitation of existing sparse streaming accelerators [2, 3, 13, 14, 22, 24, 25] is that they support only a single workload per invocation. As a result, multiple workloads must be serialized, with each request waiting for the previous one to complete. This is not suitable for environments in which multiple users must share a common hardware resource, such as data centers. A naïve solution is to deploy multiple accelerator instances, but this brute-force approach fails to address the fundamental challenge of enabling multiple workloads to run concurrently on a single accelerator. Moreover, replicating accelerators significantly increases both power consumption and system cost, making it an impractical long-term solution.

2.3 PE-aware Non-zero Scheduling

In the context of sparse streaming accelerators, scheduling data in memory directly reflects how data is scheduled in the PEs, which allows us to treat these terms interchangeably. Recent sparse streaming accelerators [13, 24, 25] use PE-aware non-zero scheduling to schedule the sparse matrix on to the off-chip memory/PEs. In PE-aware non-zero scheduling, each PE is assigned rows according to the following equation:

$$PE_{id} = row_{id} \,\%\, Total\ PEs; \quad PE[PE_{id}] \leftarrow row[row_{id}] \qquad (1)$$

However, in-order scheduling of the instructions of these rows results in read afer write (RAW) dependencies. To understand this, we analyze an example of a sparse matrix A, dense vector x and a Compute unit with two PEs (PE0 and PE1), to perform the SpMV operation ($y = Ax$). The rows scheduled for PE0 are $A_{0,0:j-1}, A_{2,0:j-1}, A_{4,0:j-1}$ and so on, where j represents the total

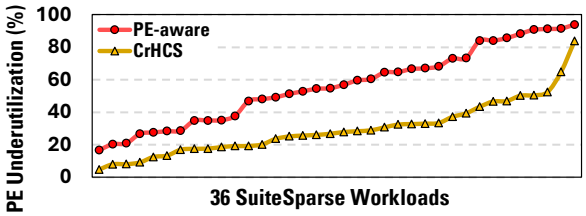


Figure 1: PE underutilization (%) of PE-aware non-zero scheduling and CrHCS on SuiteSparse workloads (lower is better). The PE underutilization is high for both the cases.

columns. The in-order data scheduling for PE0 results in the following instructions per clock cycle (cc):

cc:1 I1: $psum_0 \leftarrow A_{0,0} \times x_0$
cc:2 I2: $psum_0 \leftarrow A_{0,1} \times x_1 + psum_0$

It is evident that a RAW dependency exists between Instructions I1 and I2 because of the partial sum ($psum_0$), which indicates that I2 must wait for I1 to complete before it can execute. This introduces stalls and underutilizes PE0. PE-aware non-zero scheduling mitigates this issue by employing a cyclic row-wise assignment. Instead of processing one row sequentially before moving to the next, the PE executes instructions from other rows and later returns to the first row, hence the term "cyclic". With PE-aware non-zero scheduling, the instructions are rescheduled as follows:

cc:1 I1: $psum_0 \leftarrow A_{0,0} \times x_0$
cc:2 I2: $psum_2 \leftarrow A_{2,0} \times x_0$
cc:3 I3: $psum_4 \leftarrow A_{4,0} \times x_0$
cc:4 I4: $psum_0 \leftarrow A_{0,1} \times x_0 + psum_0$

If the RAW dependency distance is less than 3, Instruction I4 can start executing in the 4^{th} cycle without introducing any RAW dependency or stalls in the pipeline.

2.4 Cross-HBM Channel OoO Scheduling

While PE-aware non-zero scheduling reduces stalls, it does not eliminate them entirely. For instance, if no non-zero values exist in $A_{2,0:j-1}$ and $A_{4,0:j-1}$, the pipeline will experience stalls, leading to PE underutilization. The set of instructions will become:

cc:1 I1: $psum_0 \leftarrow A_{0,0} \times x_0$
cc:2 I2: Stall
cc:3 I3: Stall
cc:4 I4: $psum_0 \leftarrow A_{0,1} \times x_0 + psum_0$

To address this limitation, cross-HBM channel out-of-order (OoO) scheduling (CrHCS) [3] schedules instructions from a neighboring Compute unit on to the PEs of the current Compute unit. For instance, if two instructions, i1 and i2, are scheduled in a neighboring Compute unit, CrHCS will insert these instructions between I1 and I4 of the current Compute unit's execution schedule. As a result, the instructions assigned to PE0 of the current Compute unit will be:

cc:1 I1: $psum_0 \leftarrow A_{0,0} \times x_0$
cc:2 I2: i1 of neighboring Compute unit
cc:3 I3: i2 of neighboring Compute unit
cc:4 I4: $psum_0 \leftarrow A_{0,1} \times x_0 + psum_0$

Although CrHCS significantly reduces underutilization, it still falls short when a neighboring Compute unit runs out of instructions, causing substantial stalls in the PEs. To validate our claims

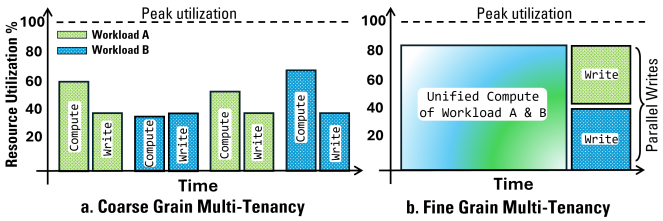


Figure 2: An example of resource utilization in coarse-grain and fine-grain multi-tenancy.

about the PE underutilization in both PE-aware non-zero scheduling and CrHCS, we ran experiments on 36 highly sparse workloads (given in Table 1) from the SuiteSparse [5] dataset. Figure 1 shows the PE underutilization of PE-aware and CrHCS. It can been seen that for both of these cases the underutilization is significantly high, leaving room for improvement.

3 Key Insights

These challenges motivate us to develop a fine-grain multi-tenancy framework that allows a single sparse streaming accelerator to support multiple workloads while ensuring minimal PE underutilization. To enable fine-grain multi-tenancy, the key insight of our work is that ***if we fuse the PE instructions of multiple workloads in to a unified set of instructions, it enables multi-tenant execution of a PE while eliminating PE stalls by filling them with instructions from other workloads.*** To achieve this we propose **Procyon**, a fine-grain multi-tenancy framework that enables sparse streaming accelerators to execute instructions from multiple workloads within a single accelerator instance while ensuring high PE utilization and performance.

4 Procyon - Scheduler

In this section, we discuss how Procyon unifies instructions associated with multiple workloads into a single execution schedule.

4.1 Level of Granularity

The primary objective of Procyon is to allow multiple workloads to share the accelerator fabric concurrently while also mitigating the PE underutilization that arises when the accelerator is dedicated to a single workload. Multi-tenancy can be implemented at different levels of granularity: *coarse-grain* or *fine-grain*. Procyon adopts fine-grain multi-tenancy based on its better control over PEs. The detailed rationale behind this choice is discussed below.

In **coarse-grain multi-tenancy**, control is managed at the device level, that is, the accelerator is fully allocated to one workload for a fixed duration before switching to the next. Although this approach allows multiple workloads to execute within a single invocation of the accelerator, it does not address PE underutilization. For example, Figure 2a shows an example of coarse-grained multi-tenancy. While multiple workloads do share the accelerator, the execution remains inefficient as there is no overlapping between the `'compute'` stages of two workloads. As a result, none of the `'compute'` stage reaches peak (or closer to peak) resource utilization. It can also be observed that the scalar stages (a stage is scalar if there are no dependencies between its different clock cycles) such as `'write'` of different workloads can not be parallelized because of the sequential nature of coarse grain multi-tenancy.

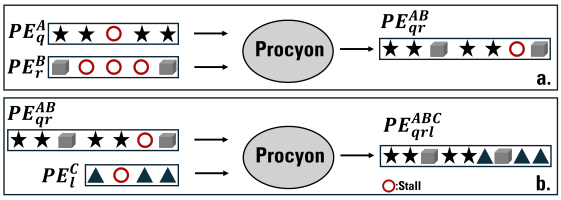


Figure 3: Example of Procyon applied to one PE from each of the three workloads

On the other hand, **fine-grain multi-tenancy** provides PE-level granularity, allowing instructions from multiple workloads to be executed together in a unified `compute` stage as shown in Figure 2b. As a result, a PE can execute instructions from multiple workloads, enabling true concurrent execution and high resource utilization. Once this unified compute stage completes, it is equivalent to having completed the compute phase of all participating workloads, thereby enabling parallel execution of next scalar stages such as `'write'` stage. Based on these reasons, Procyon adopts a fine-grain multi-tenancy to improve PE utilization.

4.2 Mechanism

Key Idea: The key idea behind enabling fine-grain multi-tenancy is to fuse the instructions of a PE from one workload with those of a PE from another workload. As a result, Procyon produces a single, unified set of instructions (execution schedule) for each PE. This allows the PE to implicitly execute instructions belonging to different workloads without requiring explicit context switching, essentially making it a multi-tenant PE.

Working: As discussed in Section 1, the way data is scheduled on to the memory directly determines how the instructions are mapped on to the PEs. In the datalist scheduled on memory, a zero represents a stall cycle in the PE. Building on this principle, Procyon enables fine-grain multi-tenancy by fusing and rescheduling the datalists associated with each PE. The mechanism of Procyon is straightforward: Procyon replaces stalls in the datalist of one PE with available non-zero entries from the datalist of another workload's PE, thereby reducing idle cycles and improving utilization.

Example: Figure 3 shows an example of Procyon applied on three workloads, that is, Workload A, B, and C. Figure 3(a) shows the datalists of q^{th} and r^{th} PE of Workload A and B, that is, PE_q^A and PE_r^B, respectively. Each position in the list corresponds to one execution cycle on the PE. Procyon fuses these two lists by filling the stall cycles in PE_q^A with the available non-zero entries from PE_r^B producing a unified datalist PE_{qr}^{AB}. Note that this fused list may still contain stalls. These remaining stalls are required to preserve the RAW dependencies present within the migrated sequence from PE_r^B. Specifically, in this example, the datalist of Workload B contains three cycles between its first and last non-zero values. To preserve the RAW dependencies across these entries, Procyon must maintain at least the same spacing in the unified datalist. This is why a stall remains in PE_{qr}^{AB} even after filling all other available stalls with PE_r^B non-zeros (cubes in Figure 3). Figure 3(b) applies Procyon again to increase the scope of multi-tenancy from two to three workloads. This time Procyon merges the unfied datalist PE_{qr}^{AB} with datalist of PE_l^C. The resulting unfied datalist, PE_{qrl}^{ABC}, contains no remaining

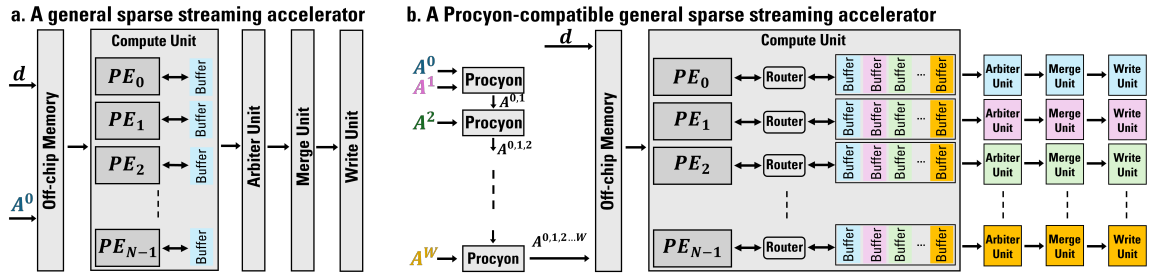


Figure 4: Procyon allows multiple workloads to execute without changing the number of PEs in the Compute unit. It also parallelizes scalar units (e.g., Arbiter unit). A^W stands for W^{th} workload. d stands for the dense operand.

stalls. Note that, PE_{qrl}^{ABC} is a single fine-grain multi-tenant PE, not a group of three PEs. It can execute instructions of q^{th}, r^{th}, and l^{th} PEs of Workload A,B and C respectively.

Degree of Multi-Tenancy: Procyon does not impose a fixed limit on the number of tenants that can be mapped onto a single PE. The degree of multi-tenancy is determined entirely by architectural policy and available resources. System designers may choose to cap the number of supported workloads based on deployment requirements, but Procyon itself can process any number of tenants as long as the underlying hardware permits.

4.3 PE Pairing

Procyon schedules the datalist of each PE to enable fine-grain control over the PEs. This ability enables Procyon to flexibly decide which PE from one workload should be fused with which PE from another workload. In other words, it can control the PE pairing strategy and switch between them when required. Procyon can employ one of these approaches to make this pairing decision:

4.3.1 1:1 PE Pairing. In this scheme, each PE from Workload A is paired with the PE of the same index from Workload B:

$$PE_n^{(A)} \longleftrightarrow PE_n^{(B)} \quad \forall\, n \in \{0, 1, \ldots, N-1\}. \tag{2}$$

N is the total number of PEs. This pairing is simple and easy to implement, but it does not necessarily yield optimal utilization as it may not align with the sparsity patterns of the two workloads.

4.3.2 Greedy PE Pairing. In this scheme, $PE_n^{(A)}$ from Workload A is paired with the PE from Workload B that results in the minimum number of stalls when fused together. Formally,

$$PE_m^{(B)} = \arg\min_k \text{Stalls}\big(\text{PE}_n^{(A)},\ PE_k^{(B)}\big), \tag{3}$$

where the pairing is chosen to minimize PE underutilization/stalls. This approach fixes a PE from Workload A and searches all PEs in Workload B to find the most efficient match.

4.3.3 Global PE Pairing. This approach searches across all PEs in both workloads to find the pair $(PE_n^{(A)}, PE_m^{(B)})$ that yields the globally minimum number of stalls. Formally,

$$(n^*, m^*) = \arg\min_{n,m} \text{Stalls}\big(\text{PE}_n^{(A)},\ PE_m^{(B)}\big). \tag{4}$$

The selected pair (n^*, m^*) satisfies:

$$\text{Stalls}\big(PE_{n^*}^{(A)}, PE_{m^*}^{(B)}\big) \le \text{Stalls}\big(PE_{n^*}^{(A)}, PE_k^{(B)}\big), \qquad \forall k, \tag{5}$$

$$\text{Stalls}\big(PE_{n^*}^{(A)}, PE_{m^*}^{(B)}\big) \le \text{Stalls}\big(PE_j^{(A)}, PE_{m^*}^{(B)}\big), \qquad \forall j. \tag{6}$$

For N number of PEs, although the search space increases to $O(N^2)$, this method provides the most optimal pairing and gives minimum possible PE underutilization.

5 Procyon - Architectural Support

The benefits of multi-tenancy can only be fully realized if running multiple workloads does not introduce additional execution time overhead to execute scalar operations on the outputs of different workloads. Procyon enables this by employing multi-tenancy at the level of each PE (hence, the term *fine-grain multi-tenancy*). This allows the workloads to be computed together without employing new PEs and enabling scalar functions, for example, arbitration, merge, and write-back, to operate in parallel for each workload.

Figure 4a shows a general high-level architecture of sparse streaming accelerators [3, 13, 24, 25]. These accelerators are designed for a single sparse workload A^0 and their Compute unit only does $A^0 \times d$ for a single workload across N number of PEs, where d is the dense operand. The Compute unit forwards its output to the Arbitration and Merge logic where the data is rearranged and written back to the off-chip memory. To run multiple workloads concurrently, multiple instances of the same accelerator has to be instantiated owing to their single workload execution model.

Figure 4b shows the architectural additions required to support Procyon. Procyon receives the datalists from multiple workloads and produces a unified datalist which is then mapped on to the PEs in the Compute unit. Procyon does not impose any inherent limit on the number of workloads that can be fused and the choice is governed entirely by the hardware resources allocated to the accelerator as discussed in Section 4.2. As shown in Figure 4b, the Compute unit still contains N PEs, identical to the baseline design in Figure 4a. However, with Procyon, these N PEs are utilized far more effectively, as they now execute operations from multiple workloads simultaneously, performing $A^0 \times d$, $A^1 \times d$, ... $A^W \times d$.

As discussed in Section 4.1, one of the key advantages of fine-grain multi-tenancy is that it enables a unified `compute` stage while still allowing each workload to execute its scalar stages (e.g., such as the `write` stage) in parallel. To support this capability, the architecture must be able to distinguish the outputs belonging to different workloads. Procyon accomplishes this by tagging every value in the unified datalist with an identifier indicating its source workload. For example, non-zero entries from A^0 are tagged with 0, those from A^1 with 1, and so on. The Compute unit receives these tags and uses the Router unit to route each output value to the corresponding on-chip buffers associated with its workload. Once the Compute stage produces these tagged outputs, the results of each workload

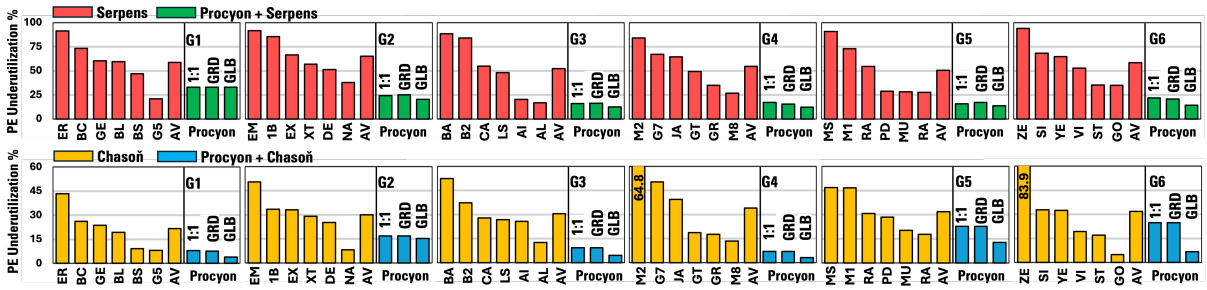


Figure 5: PE underutilization (%) of Serpens [24] and Chasoň [3] with and without Procyon on SuiteSparse workloads [5] (lower is better). AV stands for Average.

Table 1: SuiteSparse [5] workloads divided into 6 groups.

Group	Workload	Tag	Sp.%	Group	Workload	Tag	Sp.%
G1	eris1176	ER	1.34	G2	email	EM	0.84
	bcspwr08	BC	0.22		1138_bus	1B	0.31
	gemat12	GE	0.13		ex3	EX	1.58
	blckhole	BL	0.32		extr1b	XT	0.13
	bcsstk23	BS	0.45		delaunay_n	DE	0.29
	G55	G5	0.09		nasa4704	NA	0.47
G3	bayer06	BA	0.22	G4	M20PI_n1	M2	0.20
	b2_ss	B2	0.32		g7jac020	G7	0.12
	cavity13	CA	1.06		jan99jac020	JA	0.073
	lshp2614	LS	0.263		G30	GT	0.90
	airfoil1_du	AI	0.036		grid2_dual	GR	0.12
	Alembar	AL	0.10		M80PI_n1	M8	0.061
G5	msc01050	MS	2.37	G6	zenios	ZE	0.015
	mice_10N1	MI	1.16		SiH4	SI	0.67
	rajat03	RA	0.056		yeast_30N1	YE	2.38
	Pd	PD	0.019		viscoplastic1	VI	0.038
	Muu	MU	0.33		struct4	ST	2.16
	rail_5177	RA	0.13		goodwin	GO	0.60

are forwarded independently and in parallel to their respective Arbitration, Merge, and Write units. These scalar units run concurrently and do not interfere with one another, thereby preserving parallelism and preventing bottlenecks across workloads.

6 Experimental Setup

Baselines: We use two recent sparse streaming accelerators, Serpens [24] and Chasoň [3], as our baselines. Serpens employs PE-aware non-zero scheduling, while Chasoň uses CrHCS for scheduling non-zero values on off-chip memory and equivalently, on the PEs. Both accelerators use 16 512-bit wide HBM channels to transfer the input sparse matrix A to their Compute units, which contain 128 PEs running in parallel. The high-level architecture of Serpens and Chasoň are similar to the one shown in Figure 4a, with minor differences in the Compute unit and scalar units.

Simulation Infrastructure: We use the publicly available source code of both Serpens [24] and Chasoň [3]. For our experiments, we generate the bitstream for both accelerators on AMD Alveo U55C FPGA. To integrate Procyon with these baselines, we modify their C++ HLS code to ensure compatibility with Procyon's fine-grain multi-tenancy mechanisms. To evaluate the performance of Procyon, we develop a cycle-accurate simulator that uses clock cycle data from the HLS C-synthesis output. For a fair comparison, we simulate Procyon on top of both Serpens and Chasoň at the same frequencies achieved by the original FPGA implementations: Serpens operates at 249 MHz, while Chasoň runs at 301 MHz. Our simulator provides latency estimates with an accuracy of within 1% of the actual FPGA execution time. We are evaluating our work and baselines on the SpMV kernel.

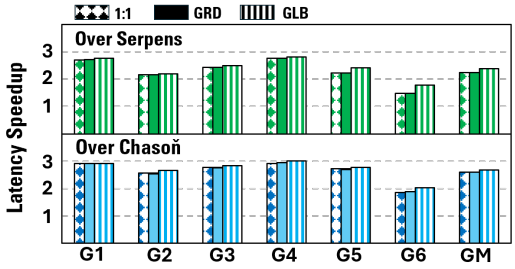


Figure 6: Latency Speedup of Procyon (with different PE pairings) vs. Serpens [24] and Chasoň [3] for the group (G) of SuiteSparse workloads [5] (higher is better). GM stands for geometric mean.

Workloads: We use 36 workloads, shown in Table 1, from the SuiteSparse dataset [5]. For evaluation, we randomly divide these workloads into six groups of six workloads each. To evaluate Procyon, all workloads in a group are executed within a single instantiation of sparse streaming accelerator. This is made possible by Procyon, which transforms the underlying sparse streaming accelerator into a fine-grain multi-tenant architecture.

7 Evaluation

7.1 PE underutilization

We experiment Procyon with different PE Pairing schemes that are described in Section 4.3, that is, 1:1 PE Pairing (1:1), Greedy PE Pairing (GRD) and Global PE Pairing (GLB). Figure 5a compares the PE underutilization of Serpens [24] with Procyon compatible Serpens. We can see that when workloads are executed individually on Serpens, the accelerator exhibits high PE underutilization. Procyon significantly reduces this underutilization by enabling fine-grain multi-tenancy across all six workloads within each workload group. We can also observe that Procyon improves the PE underutilization under all three PE pairing schemes.

Figure 5b reports the PE underutilization of Chasoň and Procyon compatible Chasoň [3]. As compared to Serpens (Figure 5a), Chasoň [3] already exhibits lower PE underutilization due to its better scheduling scheme, CrHCS, which reschedules instructions across PE-aware scheduled PEs, but only within a single workload. Procyon further reduces this underutilization substantially. For instance, in Group G1, Chasoň [3] achieves an average PE underutilization of 21.6%. With Procyon, this drops to 7.9% under the 1:1 PE-pairing scheme, and decreases even further to 3.8% when using Global PE Pairing. Still some PE underutilization exists in Procyon because when a workload does not have enough non-zero values to fill the stalls in the datalist of another workload.

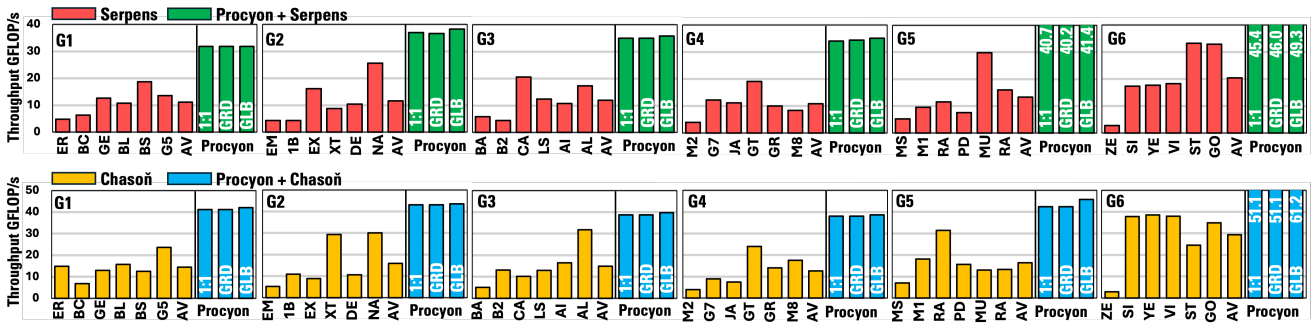


Figure 7: Throughput (GFLOP/s) of Serpens [24] and Chasoň [3] with and without Procyon on SuiteSparse workloads [5] (higher is better). AV stands for Average.

7.2 Speedup

Improvements in PE utilization result in PEs remaining active for a greater portion of the time, allowing more work to be completed within the same compute budget. To analyze these performance gains, we can analyze multiple metrics discussed below.

Latency: Figure 6 illustrates the speedup achieved by Procyon-compatible sparse streaming accelerators (Serpens and Chasoň) compared to their original implementations. For each group of workloads, we compute the average latency across all workloads running on Serpens and Chasoň, then compare it to the latency of the Procyon-compatible variants. As shown, Procyon achieves a speedup of up to 2.9× over Serpens, with an average speedup of approximately 2.3×. Similarly, Procyon delivers a speedup of up to 3× over Chasoň, with an average speedup of 2.6×.

We can also observe that Procyon performs well under all three PE pairing schemes. Notably, the improvement achieved with 1:1 PE Pairing is comparable to that of Greedy PE Pairing (GRD). As expected, Global PE Pairing (GLB) consistently delivers the best performance across all experiments, since it exhaustively searches for the optimal PE pairs (Section 4.3). However, the difference in improvement compared to 1:1 and Greedy pairing is relatively small. Therefore, in scenarios where an exhaustive $O(N^2)$ search is unnecessary (due to time constraints), 1:1 PE pairing can be employed as a lightweight alternative with $O(1)$ complexity.

Throughput: Figure 7 quantifies the throughput (GFLOP/s) of Procyon-compatible architectures and the baselines. Procyon converts a single sparse streaming accelerator into the throughput equivalent of two or more standalone Serpens or Chasoň instances without employing any additional PEs. Procyon-compatible Serpens [24] achieves up to 49.3 GFLOP/s for Group 6 (G6), compared to an average of 20 GFLOP/s in Serpens [24]. Similarly, Procyon-compatible Chasoň [3] reaches 61.2 GFLOP/s for G6, versus an average of 30 GFLOP/s for the individual workloads on Chasoň.

Bandwidth Efficiency: Serpens [24], Chasoň [3] and their Procyon-compatible variants are equipped with HBM on AMD Alveo U55C that provides $14.37GB/s$ bandwidth (BW) per channel. Each accelerator design uses 16 HBM channels to store the sparse matrix, yielding a total of $16 \times 14.37 = 229.9GB/s$ peak input bandwidth. To analyze the bandwidth usage, we use *Bandwidth Efficiency*, defined as $\frac{Throughput}{Peak\ Input\ B/W}$ in [3, 24], as a metric. Figure 8 shows the bandwidth efficiency of Serpens [24], Chasoň [3] and their Procyon-compatible variants. Procyon achieves better bandwidth efficiency across all the workload groups.

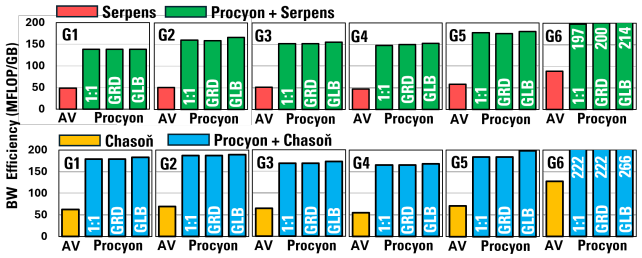


Figure 8: Bandwidth (BW) efficiency of Serpens [24] and Chasoň [3] with and without Procyon for six group (G) of SuiteSparse workloads [5] (higher is better). AV stands for Average.

7.3 Monetary Cost Analysis

A major advantage of enabling multi-tenancy is eliminating the need to instantiate multiple accelerator copies to serve concurrent workloads. With Procyon, several workloads can execute simultaneously on a single sparse streaming accelerator. This not only improves performance and PE utilization but also leads to substantial reductions in system cost and operational overhead. In our evaluation, each group contains six workloads. In a traditional setting, supporting the concurrent execution of these six workloads would require six separate accelerator instances, that is, six instances of Serpens [24] or Chasoň [3]. Assuming deployment on AMD's Alveo U55C platform, this configuration would require six FPGA cards. At a list price of 4,747 USD per device [30], the total hardware cost amounts to 28,482 USD. In comparison, Procyon enables the same six workloads to run concurrently on a single Alveo U55C board, reducing the hardware cost by a factor of 6×. Furthermore, because only one FPGA is required instead of six, the overall power consumption is also reduced by an order of magnitude, resulting in both economic and energy-efficiency benefits.

8 Conclusions

Existing sparse streaming accelerators remain limited by single-workload execution and substantial PE underutilization caused by non-zero scheduling inefficiencies. To address these limitations, we proposed Procyon, a fine-grain multi-tenancy framework that enables a single instance of sparse streaming accelerator to execute multiple workloads concurrently. Our evaluation shows substantial improvement in PE utilization and performance (throughput, BW efficiency, etc), making Procyon a cost-effective solution.

Acknowledgment

Procyon is supported by DoE-ASCR Award DE-SC0024079.

References

[1] Ubaid Bakhtiar, Helya Hosseini, and Bahar Asgari. 2024. Acamar: A Dynamically Reconfigurable Scientific Computing Accelerator for Robust Convergence and Minimal Resource Underutilization. In *2024 57th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 1601–1616. doi:10.1109/MICRO61859.2024.00117

[2] Ubaid Bakhtiar, Donghyeon Joo, and Bahar Asgari. 2025. Pipirima: Predicting Patterns in Sparsity to Accelerate Matrix Algebra. In *2025 62nd ACM/IEEE Design Automation Conference (DAC)*. 1–7. doi:10.1109/DAC63849.2025.11132628

[3] Ubaid Bakhtiar, Amirmahdi Namjoo, and Bahar Asgari. 2025. Chasoň: Supporting Cross HBM Channel Data Migration to Enable Efficient Sparse Algebraic Acceleration. In *Proceedings of the 58th IEEE/ACM International Symposium on Microarchitecture (MICRO '25)*. Association for Computing Machinery, New York, NY, USA, 778–794. doi:10.1145/3725843.3756086

[4] Hana Chitsaz, Johnson Umeike, Amirmahdi Namjoo, Babak N. Safa, and Bahar Asgari. 2025. Belenos: Bottleneck Evaluation to Link Biomechanics to Novel Computing Optimizations. arXiv:2510.15908 [cs.AR] https://arxiv.org/abs/2510.15908

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

[6] Michael deLorimier, Nachiket Kapre, Nikil Mehta, Dominic Rizzo, Ian Eslick, Raphael Rubin, Tomas E. Uribe, Thomas F. Jr. Knight, and Andre DeHon. 2006. GraphStep: A System Architecture for Sparse-Graph Algorithms. In *2006 14th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*. 143–151. doi:10.1109/FCCM.2006.45

[7] Ruibo Fan, Xiangrui Yu, Peijie Dong, Zeyu Li, Gu Gong, Qiang Wang, Wei Wang, and Xiaowen Chu. 2025. SpInfer: Leveraging Low-Level Sparsity for Efficient Large Language Model Inference on GPUs. In *Proceedings of the Twentieth European Conference on Computer Systems* (Rotterdam, Netherlands) *(EuroSys '25)*. Association for Computing Machinery, New York, NY, USA, 243–260. doi:10.1145/3689031.3717481

[8] Boyuan Feng, Yuke Wang, Guoyang Chen, Weifeng Zhang, Yuan Xie, and Yufei Ding. 2021. EGEMM-TC: accelerating scientific computing on tensor cores with extended precision. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*.

[9] Sukhpal Singh Gill, Huaming Wu, Panos Patros, Carlo Ottaviani, Priyansh Arora, Victor Casamayor Pujol, David Haunschild, Ajith Kumar Parlikad, Oktay Cetinkaya, Hanan Lutfiyya, Vlado Stankovski, Ruidong Li, Yuemin Ding, Junaid Qadir, Ajith Abraham, Soumya K. Ghosh, Houbing Herbert Song, Rizos Sakellariou, Omer Rana, Joel J.P.C. Rodrigues, Salil S. Kanhere, Schahram Dustdar, Steve Uhlig, Kotagiri Ramamohanarao, and Rajkumar Buyya. 2024. Modern computing: Vision and challenges. *Telematics and Informatics Reports* 13 (2024), 100116. doi:10.1016/j.teler.2024.100116

[10] Courtney Golden, Axel Feldmann, Joel Emer, and Daniel Sanchez. 2025. Quartz: A Reconfigurable, Distributed-Memory Accelerator for Sparse Applications. In *Proceedings of the 58th IEEE/ACM International Symposium on Microarchitecture (MICRO '25)*. Association for Computing Machinery, New York, NY, USA, 929–943. doi:10.1145/3725843.3756035

[11] Ashish Gondimalla, Noah Chesnut, Mithuna Thottethodi, and TN Vijaykumar. 2019. SparTen: A sparse tensor accelerator for convolutional neural networks. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. 151–165.

[12] Kathleen E. Hamilton, Catherine D. Schuman, Steven R. Young, Ryan S. Bennink, Neena Imam, and Travis S. Humble. 2020. Accelerating Scientific Computing in the Post-Moore's Era. *ACM Trans. Parallel Comput.* 7, 1 (2020). doi:10.1145/3380940

[13] Zifan He, Linghao Song, Robert F. Lucas, and Jason Cong. 2024. LevelST: Stream-based Accelerator for Sparse Triangular Solver. In *Proceedings of the 2024 ACM/SIGDA International Symposium on Field Programmable Gate Arrays* (Monterey, CA, USA) *(FPGA '24)*. Association for Computing Machinery, New York, NY, USA, 67–77. doi:10.1145/3626202.3637568

[14] Kartik Hedge, Hadi Asghari, Michael Pellauer, Neal Crago, Aamer Jaleel, Edgar Solomonik, Joel Emer, and Christopher W. Flectcher. 2019. ExTensor: An Accelerator for Sparse Tensor ALgebra. In *2019 International Symposium on Microarchitecture (MICRO-52)*. IEEE/ACM, 319–333.

[15] Vasant G. Honavar, Mark D. Hill, and Katherine Yelick. 2016. Accelerating Science: A Computing Research Agenda. arXiv:1604.02006 [cs.CY] https://arxiv.org/abs/1604.02006

[16] Helya Hosseini, Ubaid Bakhtiar, Donghyeon Joo, and Bahar Asgari. 2025. Segin: Synergistically Enabling Fine-Grained Multi-Tenant and Resource Optimized SpMV. *IEEE Computer Architecture Letters* 24, 1 (2025), 181–184. doi:10.1109/LCA.2025.3562120

[17] Donghyeon Joo, Helya Hosseini, Ramyad Hadidi, and Bahar Asgari. 2025. Coruscant: Co-Designing GPU Kernel and Sparse Tensor Core to Advocate Unstructured Sparsity in Efficient LLM Inference. In *Proceedings of the 58th IEEE/ACM International Symposium on Microarchitecture (MICRO '25)*. Association for Computing Machinery, New York, NY, USA, 232–245. doi:10.1145/3725843.3756065

[18] Donghyeon Joo, Helya Hosseini, Ramyad Hadidi, and Bahar Asgari. 2025. Mustafar: Promoting Unstructured Sparsity for KV Cache Pruning in LLM Inference. arXiv:2505.22913 [cs.LG] https://arxiv.org/abs/2505.22913

[19] Nachiket Kapre. 2015. Custom FPGA-based soft-processors for sparse graph acceleration. In *2015 IEEE 26th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*. 9–16. doi:10.1109/ASAP.2015.7245698

[20] Zirui Liu, Chen Shengyuan, Kaixiong Zhou, Daochen Zha, Xiao Huang, and Xia Hu. 2023. RSC: Accelerate Graph Neural Networks Training via Randomized Sparse Computations. In *Proceedings of the 40th International Conference on Machine Learning (Proceedings of Machine Learning Research, Vol. 202)*, Andreas Krause, Emma Brunskill, Kyunghyun Cho, Barbara Engelhardt, Sivan Sabato, and Jonathan Scarlett (Eds.). PMLR, 21951–21968. https://proceedings.mlr.press/v202/liu23ad.html

[21] Liqiang Lu, Yicheng Jin, Hangrui Bi, Zizhang Luo, Peng Li, Tao Wang, and Yun Liang. 2021. Sanger: A co-design framework for enabling sparse attention using reconfigurable architecture. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*. 977–991.

[22] Manoj B. Rajashekar, Xingyu Tian, and Zhenman Fang. 2024. HiSpMV: Hybrid Row Distribution and Vector Buffering for Imbalanced SpMV Acceleration on FPGAs. In *Proceedings of the 2024 ACM/SIGDA International Symposium on Field Programmable Gate Arrays* (Monterey, CA, USA) *(FPGA '24)*. Association for Computing Machinery, New York, NY, USA, 154–164. doi:10.1145/3626202.3637557

[23] Elaheh Sadredini, Reza Rahimi, Mohsen Imani, and Kevin Skadron. 2021. Sunder: Enabling low-overhead and scalable near-data pattern matching acceleration. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*. 311–323.

[24] Linghao Song, Yuze Chi, Licheng Guo, and Jason Cong. 2022. Serpens: a high bandwidth memory based accelerator for general-purpose sparse matrix-vector multiplication. In *Proceedings of the 59th ACM/IEEE Design Automation Conference* (San Francisco, California) *(DAC '22)*. Association for Computing Machinery, New York, NY, USA, 211–216. doi:10.1145/3489517.3530420

[25] Linghao Song, Yuze Chi, Atefeh Sohrabizadeh, Young-kyu Choi, Jason Lau, and Jason Cong. 2022. Sextans: A Streaming Accelerator for General-Purpose Sparse-Matrix Dense-Matrix Multiplication. In *Proceedings of the 2022 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays* (Virtual Event, USA) *(FPGA '22)*. Association for Computing Machinery, New York, NY, USA, 65–77. doi:10.1145/3490422.3502357

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

[27] Nitish Srivastava, Hanchen Jin, Shaden Smith, Hongbo Rong, David Albonesi, and Zhiru Zhang. 2020. Tensaurus: A versatile accelerator for mixed sparse-dense tensor computations. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 689–702.

[28] John Towns, Timothy Cockerill, Maytal Dahan, Ian Foster, Kelly Gaither, Andrew Grimshaw, Victor Hazlewood, Scott Lathrop, Dave Lifka, Gregory D. Peterson, Ralph Roskies, J. Ray Scott, and Nancy Wilkins-Diehr. 2014. XSEDE: Accelerating Scientific Discovery. *Computing in Science Engineering* 16, 5 (2014), 62–74. doi:10.1109/MCSE.2014.80

[29] Hanrui Wang, Zhekai Zhang, and Song Han. 2021. Spatten: Efficient sparse attention architecture with cascade token and head pruning. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 97–110.

[30] Web. 2025. https://www.amd.com/en/products/accelerators/alveo/u55c/a-u55c-p00g-pq-g.html

[31] Sanjali Yadav and Bahar Asgari. 2025. Bootes: Boosting the Efficiency of Sparse Accelerators Using Spectral Clustering. In *Proceedings of the 58th IEEE/ACM International Symposium on Microarchitecture (MICRO '25)*. Association for Computing Machinery, New York, NY, USA, 809–823. doi:10.1145/3725843.3756125

[32] Sanjali Yadav, Amirmahdi Namjoo, and Bahar Asgari. 2025. Misam: Machine Learning Assisted Dataflow Selection in Accelerators for Sparse Matrix Multiplication. In *Proceedings of the 58th IEEE/ACM International Symposium on Microarchitecture (MICRO '25)*. Association for Computing Machinery, New York, NY, USA, 824–838. doi:10.1145/3725843.3756126

[33] Amir Yazdanbakhsh, Ashkan Moradifirouzabadi, Zheng Li, and Mingu Kang. 2022. Sparse Attention Acceleration with Synergistic In-Memory Pruning and On-Chip Recomputation. In *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 744–762.