A Data-Driven Dynamic Execution Orchestration Architecture

Zhenyu Bai*
zhenyu.bai@nus.edu.sg
School of Computing,
National University of Singapore
Singapore

Pranav Dangi*
dangi@comp.nus.edu.sg
School of Computing,
National University of Singapore
Singapore

Rohan Juneja
rohan@comp.nus.edu.sg
School of Computing,
National University of Singapore
Singapore

Zhaoying Li†
zhaoying@comp.nus.edu.sg
School of Computing,
National University of Singapore
Singapore

Zhanglu Yan
zlyan@nus.edu.sg
School of Computing,
National University of Singapore
Singapore

Huiying Lan
huiying.lan@lumai.co.uk
Lumai Ltd.
Oxford, UK

Tulika Mitra
tulika@comp.nus.edu.sg
School of Computing,
National University of Singapore
Singapore

Abstract

Domain-specific accelerators deliver exceptional performance on their target workloads through fabrication-time orchestrated datapaths. However, such specialized architectures often exhibit performance fragility when exposed to new kernels or irregular input patterns. In contrast, programmable architectures like FPGAs, CGRAs, and GPUs rely on compile-time orchestration to support a broader range of applications; but they are typically less efficient under irregular or sparse data. Pushing the boundaries of programmable architectures requires designs that can achieve efficiency and high-performance on par with specialized accelerators while retaining the agility of general-purpose architectures.

We introduce *Canon*, a parallel architecture that bridges the gap between specialized and general purpose architectures. *Canon* exploits data-level and instruction-level parallelism through its novel design. First, it employs a novel dynamic data-driven orchestration mechanism using programmable Finite State Machines (FSMs). These FSMs are programmed at compile time to encode high-level dataflow per state and translate incoming meta-information (e.g., sparse coordinates) into control instructions at runtime. Second, *Canon* introduces a time-lapsed SIMD execution in which instructions are issued across a row of processing elements over several cycles, creating a staggered pipelined execution. These innovations amortize control overhead, allowing dynamic instruction changes while constructing a continuously evolving dataflow that maximizes parallelism. Experimental evaluation shows that *Canon* delivers high performance across diverse data-agnostic and data-driven kernels while achieving efficiency comparable to specialized accelerators yet retaining the flexibility of a general-purpose architecture.

1 Introduction

An ideal processor would allocate nearly all its resources to computation while minimizing the cost of control and data movement. However, the end of Dennard scaling, the slowdown of Moore's law, and the disparate scaling of memory relative to logic have constrained the performance and efficiency gains of classical von Neumann architectures, moving them away from the ideal processor envisioned [16, 46]. As these architectures increasingly devote resources to control and data movement, their computational efficiency decreases. Against this backdrop, the rise of domain-specific compute-intensive applications has triggered a Cambrian explosion of novel, non–von Neumann accelerator architectures.

These domain-specific architectures incorporate specialized compute units and manage data dependencies through hardwired datapaths that *mimics* the application's intrinsic data flow [2]. This hardwiring reduces the cost of control and memory access relative to the computation. Essentially, these architectures rely on *the fabrication-time orchestration* or configuration of the compute and data dependencies. Examples include Tensor Processing Units (TPUs)[19], AI accelerators[6, 28, 48], media and protocol accelerators [20], and sparse tensor accelerators [10, 33, 41, 51], each designed solely for the respective workloads. However, when tasked with workloads beyond their intended domain, these architectures exhibit extreme *fragility* in performance [43].

In contrast, programmable architectures, such as Coarse-Grained Reconfigurable Arrays (CGRAs) and FPGAs, rely

*Both authors contributed equally to this research.
†corresponding author

on sophisticated mapping, placement, and routing software to orchestrate computation on their reconfigurable fabrics. By spatially distributing computation and communication at compile time, they achieve flexibility compared to specialized accelerators. GPUs, on the other hand, leverage an execution model with massive thread-level parallelism, relying on compilers and schedulers to manage computation. While they offer more adaptability than fixed-function accelerators, their efficiency hinges on structured and regular workload decomposition and inherent parallelism.

Irregularity is becoming an increasingly critical factor in modern workloads. For instance, sparse tensor operations in machine learning (ML) workloads bring irregularities with wide-ranging sparsity from 5% to 95%, and can appear in structured or unstructured forms, either known at compile time or determined at runtime. Supporting a broad range of sparse kernels has become essential [4, 49, 50]. Ultimately, GPUs, FPGAs, and CGRAs perform well on regular or data-parallel tasks but are less efficient for workloads with irregular dataflow and memory access patterns. While these architectures generally handle workload variations better than specialized accelerators, the compile-time orchestration and reliance on massive thread level parallelism can lead to performance degradation when faced with dynamic or unpredictable data patterns. This emphasises the need for a hybrid approach that integrates static and dynamic decision-making to handle an assortment of regular and irregular workloads with minimal resource overhead.

Contributions: We propose *Canon*[1], a novel parallel architecture that transcends traditional specialization-flexibility tradeoffs. Our objective is to push the boundaries of programmable architectures, aligning with the idea that extreme domain specialization may often be superfluous [36].

As shown in Figure 1, *Canon* is based on a 2D-mesh spatial architecture composed of Processing Elements (PEs). Unlike conventional reconfigurable architectures that rely exclusively on compile-time orchestration, *Canon* exploits a two-tier approach. ***Canon* first leverages inherent workload regularities, specifically, data-level and instruction-level parallelism, to schedule predictable high-level dataflow, and then uses dynamic scheduling in hardware via a lightweight programmable orchestrator to handle irregularities.** Therefore, the cost of dynamic scheduling and control is minimized by confining it to the irregular aspects of the workload (e.g., arbitrary input patterns), while the bulk of the execution benefits from the efficiency of regular dataflows. The orchestrator incorporates a FSM which acts as an on-the-fly data-to-instruction translator. This FSM is programmed by the compiler, which relies on static analysis of the high-level dataflow and compute organization, and at runtime, the FSM relies on input-data and neighbor messages as triggers for generating instructions.

[1] *Canon* in music stands for imitation with an offset, or a guiding principle.

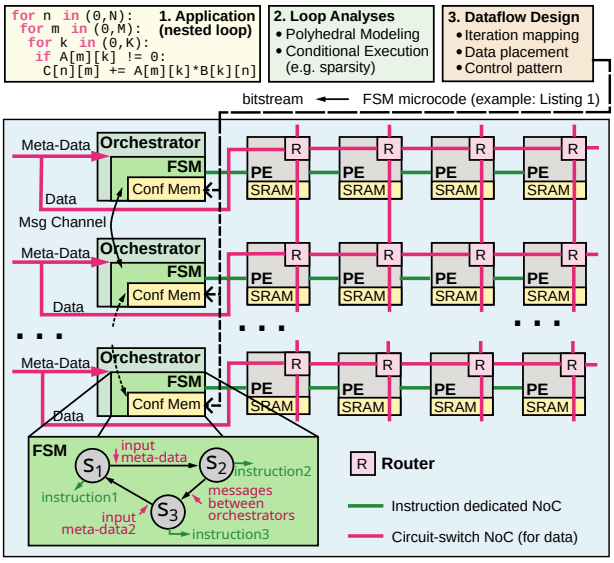


Figure 1. *Canon* Hardware Architecture

Second, *Canon* incorporates a time-lapsed SIMD execution combined with distributed memories, where instructions propagate through PEs over multiple cycles, amortizing control cost and enhancing scalability. While different PEs may be at various stages of execution at an instant, they eventually perform the same operation over time. Consequently, ***Canon* exploits an evolving dataflow across the fabric, where execution patterns are constructed, switched, and adapted over time at fine granularity to suit an unpredictable input without sacrificing parallelism.**

Control and synchronization are completely abstracted from the compute units in the PE array by resorting to orchestrators and issuing instructions in a time-lapsed manner. This abstraction enables *a priori* look-ahead into decisions, yielding predictable and deterministic behavior across the fabric, even under dynamic orchestration. The hardware sustains high parallelism, maximizes utilization, balances workload efficiently, and ultimately achieves consistently high throughput across diverse kernels and input data patterns. ***Canon* matches the performance of dense, unstructured and structured sparse accelerators in their own specialization domains with minimal efficiency degradation while also supporting a broad array of other parallel workloads typically handled by reconfigurable architectures.** To our knowledge, *Canon* is the first architecture to sustain such high performance across such a diverse set of data-agnostic and data-driven kernels.

2 *Canon* Architecture

Canon is designed to effectively handle both regular workloads and data irregularities, with the primary intention of ensuring a stable performance and minimal fragility. This section provides an overview of the architecture's design

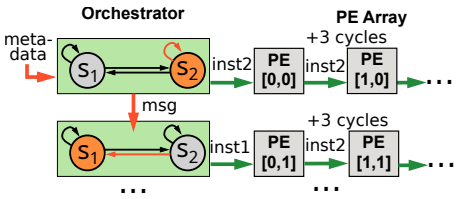


Figure 2. Orchestration and Instruction issue in *Canon*

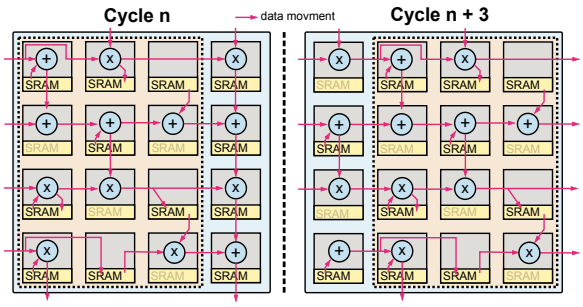


Figure 3. Time-lapsed SIMD with staggered instruction issue

principles that combine dynamic orchestration with a scalable reconfigurable fabric composed of PEs in a 2D mesh topology with a time-lapsed SIMD execution. Each PE has a vector lane for computation, associated registers, a router for communication with neighboring PEs, and a local data memory. The detailed micro-architectural implementation and mapping are further discussed in later sections.

Data-Driven Orchestration: Figure 2 illustrates the mechanism where each row of processing elements is managed by a dedicated orchestrator—a lightweight, programmable finite-state machine. A bitstream configures the FSM with states representing high-level operational modes (e.g., flushing memory or accumulating partial sums). The FSM dynamically issues instructions to its row of PEs reacting to input meta-data (e.g., sparse coordinates) and messages from neighbors. For instance, in Figure 2, both the rows are initially issued instruction 2 (inst2) corresponding to State S_2; subsequently, a message from the orchestrator at the North to South triggers the second row's transition from State S_2 to S_1, changing its output instruction from inst2 to inst1. This dynamic orchestration, which is a hybrid of compile-time mapping with runtime decision making, enables the architecture to adapt seamlessly to both regular and irregular input data patterns.

Time-Lapsed SIMD Execution: As depicted in Figure 3, the architecture employs a time-lapsed SIMD execution model wherein instructions generated by the FSM-based orchestrator propagate through the PE array over multiple cycles via a dedicated instruction network in a staggered manner. Unlike conventional SIMD execution where a single instruction is broadcast simultaneously to all PEs, here an instruction—such as "multiply inputs from SRAM and East; Send Output South"—is issued to the first PE in cycle 1, then traverses a 3-cycle pipeline before reaching the second PE in cycle 4. This staggered instruction issue ensures that while different PEs can behave differently at a given timestamp, they ultimately execute an identical sequence of operations on their respective data. The behavior, including NoC and memory actions, is replicated across the fabric. For instance, in Figure 3, the first columns' compute, memory, and NoC behavior are recreated three cycles later by the next columns.

2.1 Reduced Control Cost

An important issue with traditional von Neumann and many reconfigurable architectures is the high overhead of control logic [24, 46]. In *Canon*, orchestrators provide a lightweight control mechanism, reducing the per-PE control burden. They generate and distribute control signals via an instruction-dedicated NoC from the periphery of a PE row through staggered instruction issue. By offloading control and instruction distribution, we preserve a lightweight compute fabric, enhance scalability, simplify the programming model, and achieve high energy efficiency.

Synchronization: Each cycle, the orchestrator dispatches an instruction to the first PE of the row, which executes it and subsequently passes it along to the next PE. Consequently, every PE in a row ultimately performs the same instruction—albeit at staggered cycles and on distinct data. Each PE operates with a fixed pipeline latency of 3 cycles, ensuring that the sequence of actions initiated by the first PE (including data exchanges and memory accesses) is consistently replicated by subsequent PEs with a delay corresponding to the instruction propagation. This predictable behavior enables us to abstract inter-PE synchronization through orchestrator-level coordination. This also means effectively enhancing the compute-to-control ratio, maximizing computational density, and reducing control overhead.

Abstracted & Deterministic Irregularity Handling: The orchestrator is primarily designed to manage dynamic input variations, such as sparsity, by effectively addressing the challenges posed by irregular workloads and dynamic data dependencies. While CPUs and GPUs resolve such dependencies through memories, dataflow architectures depend largely on the NoC for data movement. In regular applications with predictable communication patterns, static routing—via circuit-switched or hardwired NoCs—is effective. However, for complex applications characterized by irregularities and variable data dependencies, packet-switched or dynamic NoCs are preferable due to their adaptability, despite incurring additional hardware overhead from mechanisms such as backpressure and virtual channels [17, 52, 53].

We design a *dynamically managed circuit-switching* scheme that handles both regular and irregular workloads with minimal overhead. Thanks to deterministic timing across the PE array from abstracted synchronization, runtime control and congestion management within the PE's NoC become unnecessary. The orchestrators manage irregularities externally, using their insight into PE determinism and input data patterns to make dynamic decisions. These decisions are embedded in the instruction stream, ensuring that the circuit-switched configurations accurately reflect the required data dependencies during execution.

2.2 Reduced Data-Handling Costs

Canon employs a distributed memory system in which each PE features local memory and communicates via the NoC. The orchestrators dynamically configure both the NoC and the memory through instructions, to efficiently support regular and irregular workloads. Although NoCs are highly efficient in data movement [16, 17], their fixed topology and limited bandwidth constrain their ability to manage complex dependencies, necessitating reliance on memory.

Canon prioritizes the NoC for mapping inherently regular compute patterns and data transfers, while it opportunistically resorts to memory when handling irregularity. In this design, memory serves a dual purpose and is partitioned respectively into two segments to prevent port saturation and minimize fragmentation: a larger segment for static data (e.g., ML weights) and a smaller scratchpad that serves as a read/write buffer to handle complex dependencies on the fly. Both segments support single-cycle random access.

As a storage device, the static data memory is mainly used for input and output data, enabling effective data reuse, reducing off-chip bandwidth requirements, and accommodating random accesses, particularly beneficial for irregular workloads with runtime-determined patterns. For resolving data dependencies, the scratchpad holds intermediate data during execution, akin to von Neumann architectures, or acts as a buffer to amortize runtime irregularities. *Canon* decouples local memory management from the compute units by integrating both software and hardware control, drawing inspiration from Explicitly Decoupled Data Orchestration (EDDO) architectures [38]. The compiler directs data distribution and programs kernel-specific data management policies into the orchestrator FSM, enabling dynamic memory management based on orchestrator instructions.

3 CANON Micro-architecture

The micro-architecture of the PE and the orchestrator is depicted in Figure 4 and Figure 5, respectively. Each PE functions as a 3-stage pipeline. At `LOAD` stage, data is loaded from the scratchpad memory, data memory, or the NoC into the vector-lane registers for input operands. The `COMPUTE` stage performs computations using a vector lane that processes four words in parallel. The results are written to the scratchpad, registers (e.g. for accumulation), data memory or sent to neighborhood PEs through the NoC at the `COMMIT` stage.

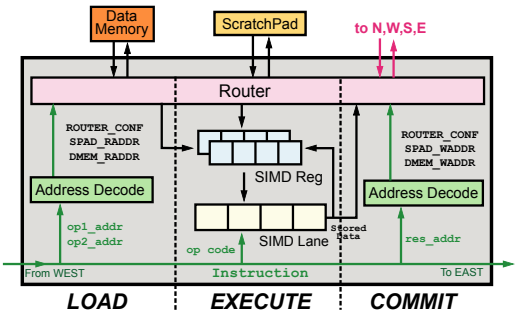


Figure 4. Architecture of 3-stage *CANON* PE pipeline

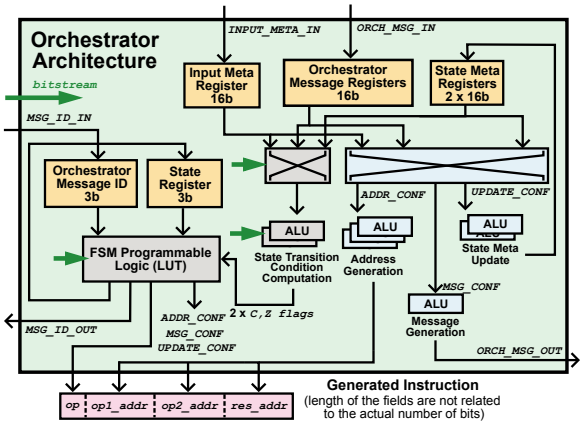


Figure 5. Architecture of the programmable Orchestrator

3.1 ISA and PE Control

PEs do not include complex control logic; they primarily rely on orchestrator-issued instructions. These instructions are streamed through the PE pipeline, dictating their execution behavior, i.e., the router, compute, and memory behavior. The instruction format is straightforward, specifying the operation, operand addresses, and destination addresses:

```
<inst> ::= <op> <op1_addr> <op2_addr> <res_addr>
```

To simplify the instruction format, the scratchpad, data memory, router, and SIMD registers share a unified address space. The specific memory accessed or NoC switching action is inferred from the address.

Since instructions are executed in a pipelined manner, hardware resource contention is minimized. The read ports of the data memory and scratchpad are accessed only during the `LOAD` stage if `op1_addr` or `op2_addr` corresponds to these memories, while write ports are used exclusively during the `COMMIT` stage if `res_addr` targets them. The NoC switch is active in both `LOAD` and `COMMIT` stages when transferring data. Due to router constraints, it supports only one data transfer

per cycle per direction. As pipeline stages are executed in parallel (ILP), a single instruction cannot simultaneously read from and write to the same NoC direction to prevent conflict with other instructions executed in parallel. This restriction is enforced at compile time.

3.2 Orchestrator Microarchitecture

The orchestrator is designed to generate instructions at runtime using its FSM, which serves as a data-to-instruction translation function. Its primary roles are to produce instruction fields, update its internal state, and send messages to neighboring orchestrators. As shown in Figure 5, the FSM's internal state is maintained using two types of registers: the *State Register*, which holds the current state, and the *State Meta Registers*, storing value-based state information such as iteration counts or memory status, depending on the target kernel. The state transition function is defined at compile time, while the state transitions are triggered at runtime by external events, including metadata from the input stream captured in the *Input Meta Register* and messages from neighboring orchestrators in the *Orchestrator Message Register* (for content) and the *Orchestrator Message ID* (for message type). We note that the semantics of the states and messages are not fixed by the hardware but instead defined by the compiler along with the definition of state transition logic.

The registers are processed for four key functions: computing the state transition condition, generating addresses, calculating the new state, and generating messages. The condition computation is statically configured (all gray components in the figure), while the other three functions are dynamically configured (blue components) by programmable logic depending on the FSM state. This programmable logic is a lookup table (LUT) unit capable of implementing any combinational logic function of its inputs. The LUT is implemented as SRAM. It contains 2^{10} entries, corresponding to all possible configurations of its 10 input bits ($2^{3+3+2\times2}$). Each entry outputs 48 bits for configuring the dynamic components, resulting in 6 KB SRAM. The dynamic control logic is defined by the programmer or compiler by specifying the encoding of the state, state meta, orchestrator messages, and message IDs. Before kernel execution, this data-instruction translation logic is prefilled into the LUT as a 'bitstream', enabling programmability of the orchestrator.

4 *Canon* Application Mapping

As *Canon* eliminates much of the control from PEs and has a new execution paradigm, it is essential to demonstrate the architecture's programmability and generalizability across various kernels. The compiler should produce a bitstream for every kernel. This bitstream programs the orchestrators and synchronizes the memory controllers to initiate the distributed data placement. The latter follows the EDDO paradigm that asynchronously moves the data between main memory and compute fabric. The current workflow leverages a combination of loop analyses and human intervention to produce the microcode for the FSM that efficiently maps arbitrary kernels. Given the vast mapping space, ensuring optimal mapping and data placement on *Canon* hardware remains an open challenge. At present, the process that compiles applications from high-level libraries and languages (e.g., PyTorch, C) is under development and falls outside the scope of this work. Instead, we focus on demonstrating the underlying logic behind the mapping in this study.

4.1 Mapping Sparse Kernels

Sparse kernels, particularly sparse tensor operations, are among the most popular and representative HPC workloads that involve input-dependent control flow. These operations create bottlenecks in applications spanning high-performance computing (HPC) and machine learning [25, 33, 41, 49, 51]. Moreover, these workloads involve varying sparse operations with different degrees of randomness or structure in inputs. Sparse tensor operations highlight two primary challenges common to applications with irregular input data. First, the sparsity in inputs introduces irregular memory access patterns, leading to memory bottlenecks. Second, the uneven distribution of non-zero elements results in workload imbalance across compute units, necessitating specialized hardware for load balancing and synchronization. Traditional architectures, including GPGPUs [8] and systolic arrays [19], often suffer from under-utilization of compute units due to this. We demonstrate the mapping of various dense and sparse kernels on *Canon* hardware.

4.1.1 Case Study: SpMM. *Canon* supports various control patterns (or dataflows). Figure 6(a) illustrates our SpMM dataflow, based on Gustavson's algorithm [10, 14, 33, 51]. In this approach, rows of the sparse matrix A are processed in parallel to produce corresponding output rows, with nonzeros streamed into the matching row of the PE array.

Each PE maintains a tile of the dense matrix B in its local memory. PEs within the same row store identical rows of B, but distinct columns. For every non-zero entry a_{ij} in a row of A, the corresponding rows of B, indexed by j, are retrieved from local memory. This enables scalar-vector multiplications between a_{ij} and the local segments of B, yielding partial sums (`psums`). These vector psums are then propagated along the PE array's column for accumulation. The final cumulative psums exit the last PE in each column, forming the resulting output matrix C. A detailed pseudo-code representation of the dataflow, obtained by restructuring (tiling, reordering) the standard triple-for loop used in matrix multiplication, is provided in Listing 3 in the Appendix. This mapping achieves exceptional efficiency in our architecture by effectively addressing **random access**, **reduction dependencies**, and **load imbalances**, while preserving high parallelism.

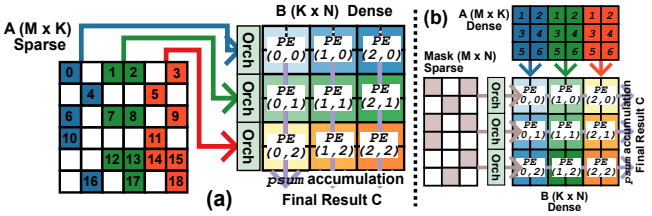


Figure 6. (a) SpMM and (b) SDDMM dataflow mapping

Local Random Accesses. We tile and partition the dense matrix B so that all PEs in the same physical row store identical rows of B, with each PE holding a distinct column segment. Consequently, when a non-zero element from A is processed, every PE in that row accesses the same address offsets in its local memory, ensuring uniform addressing, aligning with the staggered instruction issue.

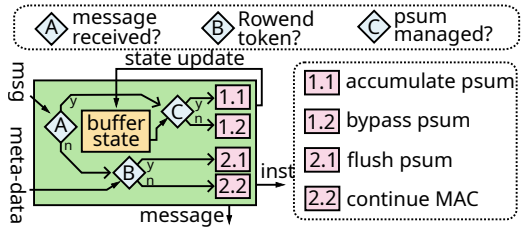


Figure 7. SpMM decision tree

Load Balancing by Asynchronous Reduction. When spatially mapping the reduction dimension (K-dimension) in matrix multiplication, a naive approach can suffer from write-after-write (WAW) hazards during the accumulation of partial sums. In SpMM kernels, non-uniform densities of non-zero elements in A may lead to load imbalance, where PEs managing denser regions are stragglers that force downstream PEs to stall, ultimately reducing overall throughput. We address this through **asynchronous reduction**. Since addition is associative, the accumulation order can be out-of-order to circumvent stalls. Each PE is concurrently responsible for two tasks: one performing multiply-and-accumulate (MAC) operations on its local non-zeros, and the other handling partial sum accumulations received from upstream PEs. Once a PE completes MAC operations for a row, it moves on to the next row without waiting for upstream PEs, thereby preventing stalls. The orchestrator dynamically determines whether a PE should execute its MAC operations or switch to accumulation tasks based on the decision tree in Figure 7. When a message is received from the north orchestrator (condition A=yes), indicating that upstream PEs are flushing their partial sums, the local orchestrator instructs the PEs to process these sums (endpoint 1.1 or 1.2). In the absence of such a message, the orchestrator directs the PEs to continue their MAC operations (endpoint 2.2).

Load Balancing by Explicit Buffer Management. Although asynchronous reduction mitigates stalls caused by WAW dependencies, it does not eliminate the underlying imbalance, so the overall execution still waits for the slowest row of PEs. To further balance the execution, each PE maintains a local scratchpad buffer to temporarily store partial sums it generates or receives, anticipating imbalance from a straggler in the array. This buffer operates on a FIFO basis, with the PE processing only those partial sums that are *managed* at a given time. The orchestrator monitors the buffer state, explicitly tracking the oldest row index. Once a PE completes processing a row of sparse elements, it flushes the oldest partial sum to the downstream PE (case 2.1). Upon receiving the partial sum, the downstream PE compares its row index against its current operating range (condition C). If the index is out of range, indicating that the current row of PEs is overloaded, the partial sum is bypassed (case 1.2), allowing the receiving PE to continue its work without interruption. Each row of PEs can thereby dynamically offload surplus work to downstream PEs, further balancing the workload.

The asynchronous reduction and explicit buffer management are programmed as microcode for the FSM, as shown in Listing 1. This microcode defines the FSM's behavior in response to events by specifying how its internal state transitions and how it generates instructions that control each PE's router, memory, and compute operations.

The performance of *Canon* in SpMM is sensitive to the scratchpad buffer size and the characteristics of the input data. A larger buffer enhances workload balancing by allowing each PE to locally store more psums, thereby alleviating congestion in downstream PEs and reducing stalls due to

Listing 1. SpMM FSM Pseudo-code

```
// NNZ: Non-Zero, CID/RID: Column/Row ID of NNZ in A
msg_from_north = {None, PSum(RID)}
input = {None, NNZ(CID), RowEnd(RID)}
buffer = (RID_start)// Buffer state
state = {MAC(RID), ACC(RID), FLUSH(RID), NOP} // FSM main State

// PE Behavior
op = MAC(CID)  if !msg_from_north && input == NNZ(CID);
     FLUSH(RID) if !msg_from_north && input == RowEnd(RID);
     ACC(RID)  if msg_from_north == PSum(RID) &&
                  buffer.is_managing(msg_from_north.RID);
     otherwise NOP;

// Memory Operations
dmem_op = LOAD[CID] if op == MAC(CID); otherwise NOP;
spad_read = LOAD[RID] if op == ACC(RID);
            LOAD[buffer.first()] if op == FLUSH(_) && buffer.is_full();
            otherwise NOP;
spad_write = FLUSH[RID] if op == FLUSH(RID) || (op == ACC(RID) && op.RID !=
       state.RID);
            otherwise NOP;

// Router Behavior
router_op = NORTH_TO_SOUTH if msg_from_north == PSum(RID) &&
       !buffer.is_managing(msg_from_north.RID) && op != FLUSH(_);
           REG_TO_SPAD if op == ACC(_) || op == FLUSH(_);
           SPAD_TO_SOUTH if op == FLUSH(_);
           SRAM_TO_REG if op == MAC(_);
           otherwise NOP;

// Buffer State Updates
buffer.RID_start += 1 if op == FLUSH(_);

// Message to south
msg_to_south = PSUM[RID] if op == FLUSH(RID) // flush
               PSUM[RID] if op == MAC(_) && msg_from_north == PSum(RID) //bypass
```

load imbalances. However, larger buffers also introduce increased overhead from more frequent buffer management operations. We evaluate the quantitative trade-offs between the scratchpad size and performance in Section 6.5.

4.1.2 Case Study: SDDMM. Sampled Dense-Dense Matrix Multiplication (SDDMM) is another important sparse-tensor kernel largely used in the sparse attention mechanisms of transformers [3, 4, 28, 48, 50]. Conceptually, SDDMM computes $C = M \cdot (A \times B)$, where the result of $A \times B$ is sampled based on a binary mask M. To exploit the sparsity in M, computations are restricted to the non-masked elements. However, SDDMM is challenging to accelerate due to the arbitrary nature of the mask, which introduces irregular data-distribution and random access patterns to the input tensors. Figure 6(b) illustrates the dataflow for SDDMM with the detailed pseudo-code in Appendix B. Since the sparsity is in the C matrix, the computation must follow an inner-product matrix multiplication dataflow. The mask M governs the computation, with its sparsity managed by orchestrators. The dense input tensor A is streamed from the top of the PE array. By the end of the PE array (the rightmost side in Figure 6(b)), a vector of V partial sums is obtained.

The proposed SDDMM dataflow also requires load balancing: elements of the A matrix are shared across PEs along the y-dimension. However, the primary source of imbalance arises from the M mask matrix, as differing numbers of non-zeros are mapped to rows processed by different PEs (i.e., PEs along the y-dimension). This imbalance complicates the sharing of A's input elements efficiently across PEs. A similar strategy than SpMM of using scratchpad to amortize the unbalance can also be applied for SDDMM, but is used to buffer the incoming A vectors to improve its reuse.

4.1.3 Case study: Structured Sparsity. Sparsity can exhibit structured patterns that are known at compile time. Common forms of structured sparsity include N:M sparsity in the input [54], i.e., N non-zeros in every M elements and diagonal-wise sparsity (commonly referred to as a window) in SDDMM operations for sparse attention [4, 18, 50]. *Canon* fully supports N:M sparsity for any N:M ratio for SpMM kernel. The global mapping is identical to SpMM, where the coordinates of sparse elements are fed to the orchestrator. With exactly N non-zero per M elements, there is no need of workload balancing with scratchpad. Instead, the psum is flushed to the next row of PEs for every N elements processed. The sliding window sparsity is similarly well-supported by *Canon* using the mapping strategy described in prior accelerators [3]. Here, the output sparsity is decomposed into dense rows, where each row corresponds to a vector-matrix multiplication. The memory can be efficiently managed for perfect data reuse between computation of rows.

4.2 Mapping General Kernels

With our primary focus on sparse tensor operations that have been showcased previously, *Canon* is able to support more general kernels. For data-agnostic kernels, i.e. where control flow remains independent of runtime data, the place-and-route-like spatial mapping techniques on reconfigurable dataflow architectures [7, 13, 23, 44] can also be applied to our architecture as show in Appendix D. However, as *Canon* has a 4-SIMD lane, for any inner loop that cannot be parallely unrolled by 4, such spatial mapping results in compute under-utilization.

More generally speaking, *Canon* can map affine loops: Let I denote the n-d iteration space of the loop nest:

$$I = \{(t_1, t_2, \ldots, s_1, s_2, \ldots) \mid t_i, s_j \in \mathbb{Z}\}, \ where$$

$t_1, t_2, \ldots$ are **temporal iterators**, meaning the iterations are executed sequentially with respect to the order among them. $s_1, s_2, \ldots$ are **spatial iterators**, meaning the iterations are executed spatially and parallely. In our case, they are only two spatial dimensions: x and y of the PE array.

Let A denote a memory accessed in the loop, to an m-d array with dimensions $d_1, d_2, \ldots, d_m$. The **access function** $f : \mathbb{Z}^n \to \mathbb{Z}^m$ maps the iteration space I to array indices $A[i_1, i_2, \ldots, i_m]$. The access function f is affine for each array dimension i_k:

$$i_k = f_k(t_1, t_2, \ldots, s_1, s_2, \ldots) = c_k + \sum_i \beta_{ki} t_i + \sum_j \alpha_{kj} s_j, \ where$$

$c_k \in \mathbb{Z}$ is a constant offset, $\beta_{ki}, \alpha_{kj} \in \mathbb{Z}^2$ are coefficients for temporal and spatial iterators, respectively.

For *Canon* to be able to share data among the neighborhood PEs with the mesh-network, the spatial iterators s_j in the access function must satisfy:

$$\exists (k, j), \alpha_{kj} \in \{-1, 0, 1\} \quad \wedge \quad \forall (k', j') \neq (k, j), \alpha_{k'j'} = 0.$$

Since finding an optimal set of loop transformations to satisfy the above conditions remains an open problem, we rely on a combination of loop analyses—primarily polyhedral-based—and human intervention to efficiently map kernels, similar to writing PTX code for GPUs [30].

5 Evaluation Methodology

We synthesize our design as configured in Table 1 using a 22nm commercial FDSOI technology node and the Synopsys Design Compiler, targeting 1GHz frequency. The design incorporates a PE array—each PE featuring a 4-wide vector lane, a router, associated SRAM as data memory and a dual-ported scratchpad—along with orchestrators positioned at the edge of each PE row. We further develop an event-driven, cycle-accurate simulator in *Rust* to provide a detailed breakdown of performance and access patterns for every architectural component when running various workloads.

Table 1. Configuration of the evaluated *Canon* Architecture

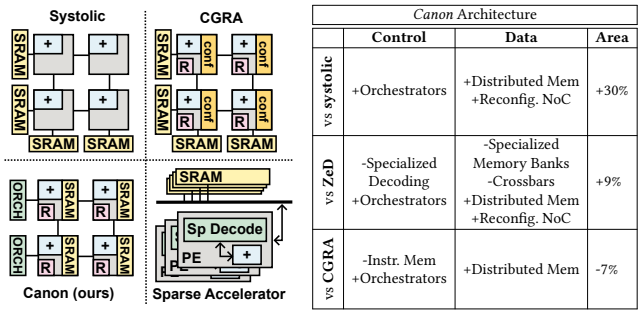
Component	Configuration
Array	8 × 8 4-SIMD INT8 array;
SRAM	4KB per PE; 288KB Overall
Scratchpad	Dual-port, 64 Bytes per PE
Orchestrator	8 orchestrators, 1 per PE row.
Main Memory	17 GB/s, LPDDR5x

Architecture. We evaluate *Canon* against four representative baselines spanning different specialization. First, the **systolic array** similar to TPU [19] serves as a reference for dense tensor accelerator. Next, the **2:4 systolic array** similar to NVIDIAs Tensor Core [29, 32] for exploiting 2:4 structured sparsity, i.e., two non-zeros in every four elements. **ZeD** [10] represents the state-of-the-art specialized sparse accelerator. Finally, a **conventional CGRA** is used to illustrate the general-purpose reconfigurable architecture.

To ensure fairness, all baselines are synthesized at the same technology node, with each architecture provisioned with an equal number of MAC units to guarantee equivalent theoretical peak compute performance. The CGRA baseline adopts a classical 2D-mesh PE architecture [12, 13, 22, 35] featuring circuit-switched NoC and a small instruction memory within each PE, sufficient for mapping the most complex kernel in our benchmarks. We use the state-of-the-art CGRA mapper [47] for complex kernels. We develop a cycle-accurate simulator to model the timing behavior of ZeD. For a fair comparison of the architecture, we exclude ZeD's preprocessing optimization of row reorganization during evaluations, as the same can be applied to *Canon*.

On-chip memory configuration significantly impacts power, area, and off-chip bandwidth requirements. For consistency, we allocate an average of 1KB of data memory per MAC unit for *Canon* and all baseline architectures. These memories are synthesized using the foundry memory compiler at the same tech node as *Canon*, with organization tailored to each architecture as shown in figure 8: *Canon* employs distributed memory local to each PE; systolic arrays and CGRAs use memory banks along the edge of the array; and the sparse accelerator follows the original design's bank organization [10]. The results are validated against the respective papers to ensure fairness of comparisons.

Workloads. We primarily utilize the sparse tensor kernels SpMM and SDDMM in ML workloads to demonstrate our architecture's resilience to input fragility. We employ the state-of-the-art sparsification technique [26] to induce sparsity in the activations of the CNN and MLP layers. The inherent sparsification in the activations leads to SpMM operations. Furthermore, we apply attention sparsification techniques from [28, 48], resulting in unstructured SDDMM operations for the QK attention matrix (later labeled SDDMM-U). We use the sliding window attention [4, 18, 49, 50](later labeled SDDMM-Win) as structured SDDMM operation. These sparsification techniques enable a trade-off between sparsity and accuracy. While high levels of sparsity (> 85-90%) may cause significant accuracy degradation depending on the model [26, 31], we conduct experiments with sparsity levels up to 95% to thoroughly evaluate our hardware's characteristics. For clarity in presenting our results, we categorize the workloads into three sparsity ranges: **S1**: Relatively dense (0–30% sparse). **S2**: Moderately sparse (30–60% sparse). **S3**: Highly sparse matrices (60–95% sparse).



Canon Architecture	Control	Data	Area
vs systolic	+Orchestrators	+Distributed Mem +Reconfig. NoC	+30%
vs ZeD	-Specialized Decoding +Orchestrators	-Specialized Memory Banks -Crossbars +Distributed Mem +Reconfig. NoC	+9%
vs CGRA	-Instr. Mem +Orchestrators	+Distributed Mem	-7%

Figure 8. Ablation of *Canon*'s features through its baselines

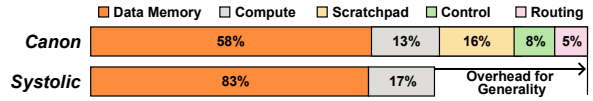


Figure 9. Area Breakdown of *Canon* and Systolic Array

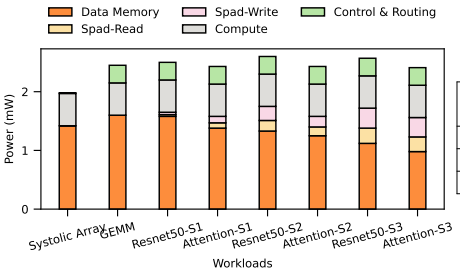
While the two sparse kernels already show the handling of different workloads, to further evaluate *Canon*, we map the kernels from the PolyBenchC [21] benchmark suite. Kernels of the suite containing square root or exponential operations in their loops are excluded due to the lack of support in both *Canon* and CGRA. The PolyBenchC kernels are further grouped into categories based on their classification within the benchmark suite, enabling a more structured analysis.

6 Evaluation Results

The baselines chosen for evaluating *Canon* cover a landscape of specialized and general architecture while simultaneously they function as effective ablation studies for its various features. Figure 8 qualitatively shows key features that can be incrementally (+) added or removed (-) from the baseline architectures to ultimately yield *Canon*. In the following sections, we quantify the impact of these features on resource consumption and performance, providing both a systematic, real-world ablation and valuable insights into the advantages of our architectural choices.

6.1 Incrementally evaluating the cost of generality

Area cost. A systolic array—representing the most densely packed 2D-mesh structure of compute units—serves as an ablation of *Canon*, wherein the routers, scratchpads, and orchestrators are omitted and the distributed memories are replaced with shared memories along the PE array edge. Figure 9 compares the area breakdown between *Canon* and



Sparsity Range	State Transitions
S1	1.94e7
S2	3.29e7
S3	9.77e7

Figure 10. Runtime power breakdown of *Canon*'s PEs (averaged), and average data-driven FSM state transitions for different sparsity ranges. Here, *Spad* stands for scratchpad.

the systolic array, showing that *Canon* incurs roughly 30% additional area, mainly due to its scratchpads, orchestrators (control), and routing, alongside a slight increase in data-memory resources for the distributed-memory design.

Circuit-switch routers account for about 5% of the PE chip area , enabling the mapping of more complex data dependencies than a rigid systolic NoC can support, similar to conventional CGRAs [12, 13, 22, 35]. For better handling irregularities, *Canon* introduces orchestrators along the array edge. These orchestrators occupy 8% of the overall area, reflecting a low control overhead. Moreover, each PE's dual-ported scratchpad contributes 13% to the chip area.

Compared to the sparse accelerator, *Canon* shows a 12% of area overhead, primarily due to the added reconfiguration capabilities and generalized memory organization. Compared to the general-purpose CGRA, benefiting from amortized control over the SIMD lane and instructions due to the orchestrators and time-lapsed execution enables *Canon* to save about 7% of total area.

Power cost. Figure 10 further illustrates the power breakdown of *Canon* across various workloads. Under GEMM, which employs a systolic-style dataflow, *Canon* consumes nearly the same power as the systolic array, with only a slight (<13%) overhead from control and routing. Notably, the GEMM power breakdown shows no dependency on scratchpads in regular applications. As input irregularity increases—from low sparsity (S1) to high sparsity (S3)—the FSM in each orchestrator triggers more data-driven state transitions to balance workload distribution. This necessitates using the scratchpads for buffering data to amortize the execution, resulting in a higher proportion of power dedicated to scratchpad operations and higher total power.

Overall, *Canon* incurs additional resource consumption relative to conventional domain-specific architectures; however, this trade-off enables it to achieve high performance across a wide range of workloads, as discussed in the following section.

6.2 Performance versus Flexibility

Figure 11 and Figure 12 quantify *Canon*'s performance relative to other architectures, serving both as a study of its fragility under low performance conditions and as an ablation of certain architectural features. *Canon* emulates the systolic dataflow of conventional systolic arrays for the GEMM kernel, exploiting kernel regularity to match their performance. However, as *Canon* allocates a portion of its resources to generality, for extremely regular, dense workloads like GEMM, a systolic array achieves a higher performance per watt, though our results indicate this performance gap is minimal. In contrast, **when workloads exhibit sparsity—which the systolic arrays dense design cannot capitalize on—their throughput can drop to less than 0.3× that of *Canon*.**

When extended to support 2:4 sparse operations [8], the modified systolic array significantly improves its performance for the corresponding 2:4 sparse structured SpMM kernel. Nonetheless, ***Canon* leverages the 2:4 structure, despite being designed agnostic to it, achieving comparable performance to the modified systolic array.** Moreover, such extreme specialization does not generalize to other input patterns or kernels, as evidenced by the modified array's diminished performance on similar (2:8 structured sparse) or dissimilar kernels (SDDDM, etc.). **In direct comparison with the sparse accelerator ZeD, *Canon* demonstrates comparable performance and efficiency on unstructured sparse kernels—the very kernels for which ZeD is specialized**. ZeD outperforms marginally (<8%) for matrices in sparsity zones S1 and select cases in S2, where specialized workload balancing through work stealing across compute units is advantageous due to a higher number of nonzeros per row. Conversely, *Canon* is better at exploiting higher sparsity levels thanks to the flexibility of its scratchpad, which mitigates imbalance and provides up to 5% better performance on some inputs. Furthermore, ZeD's fixed datapath prevent it from leveraging structured inputs, treating all matrices as unstructured and thereby missing the additional performance gains offered in N:M sparse kernels and SDDMM-Win operations. ZeD also allocates a significant portion of its power budget to address sparsity via fully connected crossbars and specialized decoders, resulting in increased power consumption that varies with the nonzero distribution pattern. In contrast, *Canon*'s homogeneous architecture opportunistically resolves load-imbalance.

SDDMM-Win1 and SDDMM-Win2 correspond to the original Longformer [4] configuration on BERT [27] (window width 512, sequence length 4K) and the Mistral-7B setup [18] (window 4K, context 16K), respectively. As the other baselines architectures lack specialization for window attention, we employ the state-of-the-art sliding chunk implementation [4] to convert the computation into multiple dense operations. **The results indicate that *Canon* outperforms**

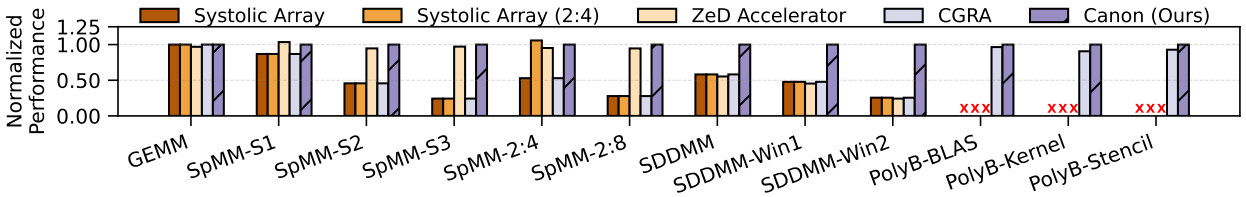


Figure 11. Speedup (fragility) of various architectures normalized to *Canon* for varying kernels and input data patterns

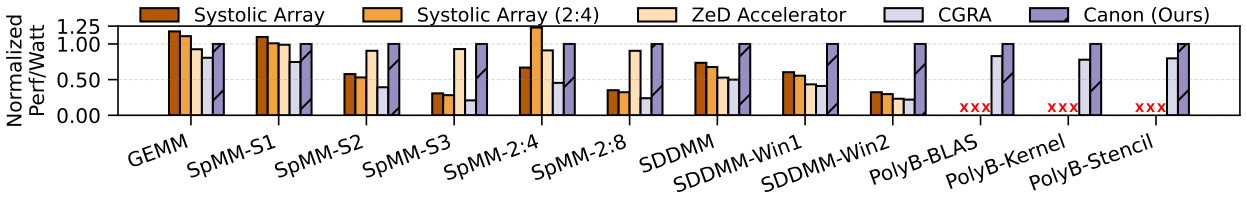


Figure 12. Performance per Watt of various architectures normalized to *Canon* for varying kernels and input data patterns

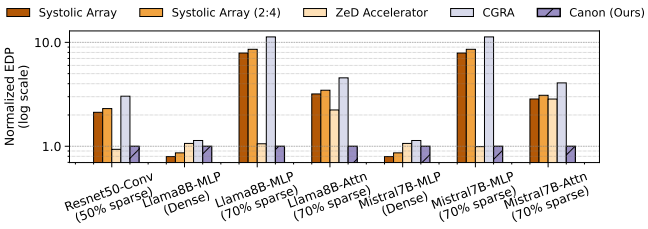


Figure 13. EDP (lower is better) of the architectures normalized to *Canon* for real ML models. (in brackets: average sparsity of the model)

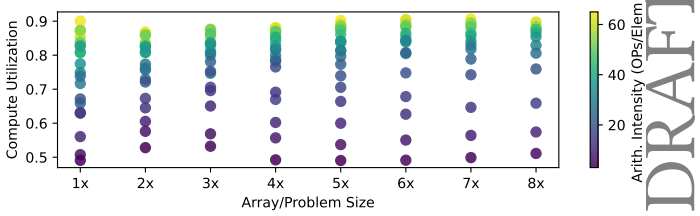


Figure 14. Sensitivity to data size & arithmetic intensity

all baselines on window attention, with a performance gains increasing at higher sparsity (Win2) due to its ability to adapt to this input pattern.

Finally, the CGRA, which can be deemed as a study for the importance of dynamic orchestration, must emulate the systolic dataflow for tensor operations since it has not dynamic mechanism to exploit sparsity, delivers performance on par with systolic arrays but at the cost of higher resource consumption, as its routing and configuration circuitry is considered overprovisioned [24]. Nevertheless, the CGRA's design prioritizes broad programmability, enabling it to execute diverse general workloads such as those in PolyBenchC. For PolyBenchC benchmarks, CGRAs outperform *Canon* in scenarios with low data parallelism, where finer-grained reconfiguration is advantageous, which constitutes some solvers in the *BLAS* set. In contrast, **when kernels exhibit sufficient parallelism—typical of most other *BLAS*, *kernel* and *stencil* workloads—*Canon* achieves better performance and power efficiency.**

Figure 13 presents a comparative evaluation of architectures on contemporary ML models through an analysis of the Energy-Delay Product (EDP). As previously discussed, *Canon* incurs a slight efficiency overhead compared to systolic arrays for entirely dense model components. However, given the diverse composition of modern models—ranging from different kernel types, such as SpMM in sparse MLPs, to a combination of SDDMM and SpMM in sparse attention, as well as varying input patterns, including unstructured sparsity in LLaMA-8B and ResNet-50 and window-structured sparsity in Mistral-7B—these results highlight the critical need for a minimally fragile architecture capable of adapting to new kernels and diverse sparsity patterns. For instance, an accelerator specialized for the moderately sparse convolutions of ResNet-50 would provide little advantage when processing a diagonally windowed sparse SDDMM operation in a model like Mistral-7B.

6.3 *Canon*'s scalability and sensitivity to data

When operating within the compute roofline, our architecture remains entirely scalable like a systolic array. By increasing the number of PEs, we can effectively scale out our architecture. However, for the irregular workloads we target, the theoretical arithmetic intensity—the number of computations per unit of data—has a significant impact on performance This overhead occurs more frequently because each data element fetched results in only a few actual computations. To assess the sensitivity of *Canon* to arithmetic intensity, we scaled both the size of the sparse tensor problems and the size of the PE array (a 8× larger workload on a 8× larger fabric). By adjusting the input shapes and sparsity levels, we acheive different arithmetic intensities for the inputs. Figure 14 illustrates the compute utilization of *Canon* with respect to problem size and arithmetic intensity. **Our results indicate that *Canon*'s compute utilization**

is primarily sensitive to arithmetic intensity, with no clear correlation to problem/array size, demonstrating the scalability of *Canon*.

6.4 Data memory Size vs. Off-Chip Bandwidth

The size of the on-chip data memory significantly influences the required off-chip bandwidth. Additionally, the bandwidth requirement depends on the arithmetic intensity of the workload. To evaluate the off-chip bandwidth requirements, we randomly generate SpMM computations across various sparsity levels, thereby altering the arithmetic intensity. We adopt a dense-stationary tiling strategy (i.e. the dense matrix stays on-chip) because the sparse input inherently reduces off-chip traffic due to its sparsity.

Figure 15 presents the results. As the arithmetic intensity decreases (i.e., as sparsity increases), our architecture maintains high throughput; however, the bandwidth requirement increases because fewer computations are performed per data element. For buffer sizes large enough to hold the entire input data (SRAM size above 576KB), the off-chip bandwidth reaches its minimal value. The increased off-chip traffic at lower arithmetic intensities reflects the extra bandwidth needed for output data. It is important to note that at higher sparsity levels, although we consume more bandwidth to maintain the same amount of computation, the effective equivalent dense computation is substantially higher. For example, at a sparsity level of 95%, we may use approximately 7x more bandwidth, but the equivalent dense throughput increases by ≈ 16x (not 20x due to under-utilization).

Considering the LPDDR5X as the off-chip memory, we plot the bandwidths for configurations using a single-die 16× lanes and a dual-die 32× module. The system-level design of *Canon* should consider the problem's arithmetic intensity and the affordable on-chip and off-chip memory devices. For instance, design point A is preferable when a higher off-chip budget is available; design point B (our current configuration) is preferable when there is a higher on-chip memory budget and a higher probability of low arithmetic intensity. If the target workload is known to have higher arithmetic intensity, one can reduce the budget for both off-chip and on-chip memory, as indicated by design point C.

6.5 Handling Load Imbalance with the Scratchpad

Figure 16 presents an ablation study highlighting the impact of scratchpad size on *Canon*'s performance. An appropriately sized scratchpad can mitigate load imbalance by buffering data and amortizing the irregularity, thereby enhancing overall fabric utilization. However, if the scratchpad is too large, it introduces overhead because PEs unnecessarily buffer data in anticipation of irregularity that may not be present. As shown in the results, incorporating an optimally sized scratchpad of 16 entries leads to 10-20% increase in compute utilization compared to 1 entry (a single register) for input sparsity levels of 60% and above.

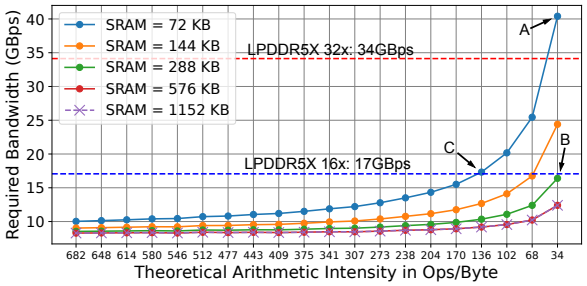


Figure 15. *Canon* bandwidth requirements to hit the compute roofline for varying arithmetic intensity (sparsity increases left to right)

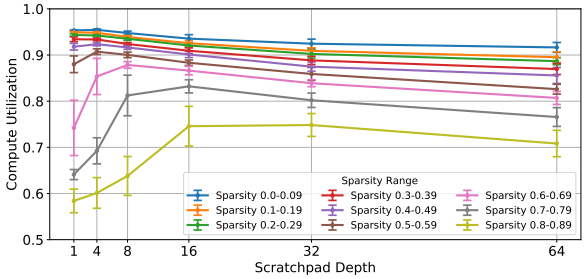


Figure 16. Impact of scratchpad depth on utilization

Although the scratchpad size is fixed at fabrication time, the effective buffer size can be software-managed by changing the memory management logic that is programmed to the orchestrator FSM. In the performance evaluations presented in Section 6.2, conservatively, we assume no prior knowledge about the sparsity and, therefore, use a buffer size of 16 entries. **By incorporating compile-time knowledge about the expected sparsity range (S1, S2, S3), *Canon* achieves an additional 5% performance improvement on average by adjusting the effective scratchpad range**

7 Related Works

Canon is fundamentally a reconfigurable architecture designed to support irregular inputs, a feature typically associated with sparse accelerators [15, 33, 41, 51]. It distinguishes itself from prior reconfigurable architectures [5, 12, 13, 22, 34, 35, 39, 42, 45] through its unique granularity of processing and reconfiguration as well as its novel approach to orchestration. When reconfigurable architectures are extended to support irregularity, it often comes with significant overhead [34]. Efforts to address irregular computation patterns frequently resort to multicore architectures [9, 37] or PEs with a much higher granularity of compute [42], as these systems can amortize the added control overhead. *Canon* fundamentally reimagines the compute on its fabric, abstracts most of its control, handling irregularity through time-lapsed execution managed by orchestrators positioned at the edge.

Industry Architectures: Commercial architectures are increasingly heterogeneous, integrating specialized units to

meet evolving workloads. With the rise of AI, most architectures now include systolic-array-based matrix multiplication units—such as Tensor Cores in NVIDIA and AMD GPUs and Google TPUs [19]. Early academic projects like Plasticine [39, 42] proposed CGRAs optimized for limited parallel patterns. However, industry adoption favored embedding systolic arrays within processing elements to support dense matrix multiplication efficiently [40], likely due to the high cost and inefficiency of reconfigurable hardware for such operations. This approach towards hardware design may reduce kernel generalizability or result in underutilized hardware, thereby negating benefits of reconfigurability [11]. Groq [1] employs a completely software-defined orchestration of heterogeneous components to emulate an array-level RISC-like processor. It demonstrates good performance on predictable, regular workloads yet suffers performance fragility on new applications or irregularity due to its reliance on extensive DLP and strictly compile-time mapping. In contrast, *Canon*'s orchestration and compute granularity minimizes fragility while maintaining high hardware utilization for moderately DLP and irregular workloads.

8 Conclusion & Future Work

We present *Canon*, a novel parallel architecture that integrates compile-time and runtime orchestration to overcome performance fragility across a diverse range of workloads. By employing FSM-based orchestration alongside time-lapsed SIMD execution, Canon leverages regular workload patterns—such as DLP and ILP—to establish a high-level dataflow while dynamically handling and adapting to irregularities. Experiments show that *Canon* achieves performance on par with accelerators in their own domains, with minimal efficiency loss, all while supporting a broader spectrum of parallel applications typically accelerated by reconfigurable architectures. Future work includes end-to-end compiler support, coming up with new techniques to exploit the architecture's programmability and dynamicity for new workloads. Canon should push the boundaries of programmable architectures, bridging the gap between ad-hoc domain specialization and general-purpose architectures.

References

[1] Dennis Abts, Jonathan Ross, Jonathan Sparling, Mark Wong-VanHaren, Max Baker, Tom Hawkins, Andrew Bell, John Thompson, Temesghen Kahsai, Garrin Kimmell, Jennifer Hwang, Rebekah Leslie-Hurd, Michael Bye, E. R. Creswick, Matthew Boyd, Mahitha Venigalla, Evan Laforge, Jon Purdy, Purushotham Kamath, Dinesh Maheshwari, Michael Beidler, Geert Rosseel, Omar Ahmad, Gleb Gagarin, Richard Czekalski, Ashay Rane, Sahil Parmar, Jeff Werner, Jim Sproch, Adrian Macias, and Brian Kurtz. 2020. Think fast: a tensor streaming processor (TSP) for accelerating deep learning workloads. In *Proceedings of the ACM/IEEE 47th Annual International Symposium on Computer Architecture* (Virtual Event) *(ISCA '20)*. IEEE Press, 145–158. https://doi.org/10.1109/ISCA45697.2020.00023

[2] Arvind and R.S. Nikhil. 1990. Executing a program on the MIT tagged-token dataflow architecture. *IEEE Trans. Comput.* 39, 3 (1990), 300–318. https://doi.org/10.1109/12.48862

[3] Zhenyu Bai, Pranav Dangi, Huize Li, and Tulika Mitra. 2024. SWAT: Scalable and Efficient Window Attention-based Transformers Acceleration on FPGAs. In *Proceedings of the 61st ACM/IEEE Design Automation Conference* (San Francisco, CA, USA) *(DAC '24)*. Association for Computing Machinery, New York, NY, USA, Article 93, 6 pages. https://doi.org/10.1145/3649329.3658488

[4] Iz Beltagy, Matthew E Peters, and Arman Cohan. 2020. Longformer: The long-document transformer. *arXiv preprint arXiv:2004.05150* (2020).

[5] Alex Carsello, Kathleen Feng, Taeyoung Kong, Kalhan Koul, Qiaoyi Liu, Jackson Melchert, Gedeon Nyengele, Maxwell Strange, Keyi Zhang, Ankita Nayak, Jeff Setter, James Thomas, Kavya Sreedhar, Po-Han Chen, Nikhil Bhagdikar, Zachary Myers, Brandon D'Agostino, Pranil Joshi, Stephen Richardson, Rick Bahr, Christopher Torng, Mark Horowitz, and Priyanka Raina. 2022. Amber: A 367 GOPS, 538 GOPS/W 16nm SoC with a Coarse-Grained Reconfigurable Array for Flexible Acceleration of Dense Linear Algebra. In *2022 IEEE Symposium on VLSI Technology and Circuits (VLSI Technology and Circuits)*. 70–71. https://doi.org/10.1109/VLSITechnologyandCir46769.2022.9830509

[6] Yu-Hsin Chen, Tushar Krishna, Joel S. Emer, and Vivienne Sze. 2017. Eyeriss: An Energy-Efficient Reconfigurable Accelerator for Deep Convolutional Neural Networks. *IEEE Journal of Solid-State Circuits* 52, 1 (2017), 127–138. https://doi.org/10.1109/JSSC.2016.2616357

[7] S. Alexander Chin, Noriaki Sakamoto, Allan Rui, Jim Zhao, Jin Hee Kim, Yuko Hara-Azumi, and Jason Anderson. 2017. CGRA-ME: A unified framework for CGRA modelling and exploration. In *2017 IEEE 28th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*. 184–189. https://doi.org/10.1109/ASAP.2017.7995277

[8] Jack Choquette, Wishwesh Gandhi, Olivier Giroux, Nick Stam, and Ronny Krashinsky. 2021. NVIDIA A100 Tensor Core GPU: Performance and Innovation. *IEEE Micro* 41, 2 (2021), 29–35. https://doi.org/10.1109/MM.2021.3061394

[9] Vidushi Dadu, Sihao Liu, and Tony Nowatzki. 2021. PolyGraph: Exposing the Value of Flexibility for Graph Processing Accelerators. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. 595–608. https://doi.org/10.1109/ISCA52012.2021.00053

[10] Pranav Dangi, Zhenyu Bai, Rohan Juneja, Dhananjaya Wijerathne, and Tulika Mitra. 2024. Zed: A generalized accelerator for variably sparse matrix computations in ml. In *Proceedings of the 2024 International Conference on Parallel Architectures and Compilation Techniques*. 246–257.

[11] Hadi Esmaeilzadeh, Emily Blem, Renee St. Amant, Karthikeyan Sankaralingam, and Doug Burger. 2011. Dark silicon and the end of multicore scaling. In *Proceedings of the 38th annual international symposium on Computer architecture*. 365–376.

[12] Graham Gobieski, Ahmet Oguz Atli, Kenneth Mai, Brandon Lucia, and Nathan Beckmann. 2021. Snafu: an ultra-low-power, energy-minimal cgra-generation framework and architecture. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 1027–1040.

[13] Graham Gobieski, Souradip Ghosh, Marijn Heule, Todd Mowry, Tony Nowatzki, Nathan Beckmann, and Brandon Lucia. 2022. Riptide: A programmable, energy-minimal dataflow compiler and architecture. In *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 546–564.

[14] Fred G. Gustavson. 1978. Two Fast Algorithms for Sparse Matrices: Multiplication and Permuted Transposition. *ACM Trans. Math. Softw.* 4, 3 (Sept. 1978), 250–269. https://doi.org/10.1145/355791.355796

[15] Kartik Hegde, Hadi Asghari-Moghaddam, Michael Pellauer, Neal Crago, Aamer Jaleel, Edgar Solomonik, Joel Emer, and Christopher W. Fletcher. 2019. ExTensor: An Accelerator for Sparse Tensor Algebra. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture* (Columbus, OH, USA) *(MICRO '52)*. Association for Computing Machinery, New York, NY, USA, 319–333. https://doi.org/10.1145/3352460.3358275

[16] Mark Horowitz. 2014. 1.1 Computing's energy problem (and what we can do about it). In *2014 IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC)*. 10–14. https://doi.org/10.1109/ISSCC.2014.6757323

[17] Natalie Enright Jerger, Tushar Krishna, and Li-Shiuan Peh. 2017. *On-chip networks*. Morgan & Claypool Publishers.

[18] Albert Q Jiang, Alexandre Sablayrolles, Arthur Mensch, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Florian Bressand, Gianna Lengyel, Guillaume Lample, Lucile Saulnier, et al. 2023. Mistral 7B. *arXiv preprint arXiv:2310.06825* (2023).

[19] Norman P. Jouppi, George Kurian, Sheng Li, Peter Ma, Rahul Nagarajan, Lifeng Nai, Nishant Patil, Suvinay Subramanian, Andy Swing, Brian Towles, Cliff Young, Xiang Zhou, Zongwei Zhou, and David Patterson. 2023. TPU v4: An Optically Reconfigurable Supercomputer for Machine Learning with Hardware Support for Embeddings. arXiv:2304.01433 [cs.AR] https://arxiv.org/abs/2304.01433

[20] Sagar Karandikar, Chris Leary, Chris Kennelly, Jerry Zhao, Dinesh Parimi, Borivoje Nikolic, Krste Asanovic, and Parthasarathy Ranganathan. 2021. A Hardware Accelerator for Protocol Buffers. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture* (Virtual Event, Greece) *(MICRO '21)*. Association for Computing Machinery, New York, NY, USA, 462–478. https://doi.org/10.1145/3466752.3480051

[21] Jeyhun Karimov, Tilmann Rabl, and Volker Markl. 2018. Polybench: The first benchmark for polystores. In *Technology Conference on Performance Evaluation and Benchmarking*. Springer, 24–41.

[22] Manupa Karunaratne, Aditi Kulkarni Mohite, Tulika Mitra, and Li-Shiuan Peh. 2017. HyCUBE: A CGRA with reconfigurable single-cycle multi-hop interconnect. In *Proceedings of the 54th Annual Design Automation Conference 2017*. 1–6.

[23] Zhaoying Li, Dan Wu, Dhananjaya Wijerathne, and Tulika Mitra. 2022. LISA: Graph Neural Network based Portable Mapping on Spatial Accelerators. In *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. 444–459. https://doi.org/10.1109/HPCA53966.2022.00040

[24] Zhaoying Li, Chenyang Yin, Thilini Kaushalya Bandara, Rohan Juneja, Cheng Tan, Zhenyu Bai, and Tulika Mitra. 2024. Enhancing CGRA Efficiency Through Aligned Compute and Communication Provisioning. *arXiv preprint arXiv:2412.08137* (2024).

[25] Aixin Liu, Bei Feng, Bing Xue, Bingxuan Wang, Bochao Wu, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, et al. 2024. Deepseek-v3 technical report. *arXiv preprint arXiv:2412.19437* (2024).

[26] James Liu, Pragaash Ponnusamy, Tianle Cai, Han Guo, Yoon Kim, and Ben Athiwaratkun. 2024. Training-free activation sparsity in large language models. *arXiv preprint arXiv:2408.14690* (2024).

[27] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. 2019. Roberta: A robustly optimized bert pretraining approach. *arXiv preprint arXiv:1907.11692* (2019).

[28] 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* (Virtual Event, Greece) *(MICRO '21)*. Association for Computing Machinery, New York, NY, USA, 977–991. https://doi.org/10.1145/3466752.3480125

[29] Stefano Markidis, Steven Wei Der Chien, Erwin Laure, Ivy Bo Peng, and Jeffrey S Vetter. 2018. Nvidia tensor core programmability, performance & precision. In *2018 IEEE international parallel and distributed processing symposium workshops (IPDPSW)*. IEEE, 522–531.

[30] Puya Memarzia and Farshad Khunjush. 2015. An In-depth Study on the Performance Impact of CUDA, OpenCL, and PTX Code. *J. Inf. Comput. Sci* 10, 2 (2015), 124–136.

[31] Iman Mirzadeh, Keivan Alizadeh, Sachin Mehta, Carlo C Del Mundo, Oncel Tuzel, Golnoosh Samei, Mohammad Rastegari, and Mehrdad Farajtabar. 2023. Relu strikes back: Exploiting activation sparsity in large language models. *arXiv preprint arXiv:2310.04564* (2023).

[32] Asit Mishra, Jorge Albericio Latorre, Jeff Pool, Darko Stosic, Dusan Stosic, Ganesh Venkatesh, Chong Yu, and Paulius Micikevicius. 2021. Accelerating Sparse Deep Neural Networks. arXiv:2104.08378 [cs.LG] https://arxiv.org/abs/2104.08378

[33] Francisco Muñoz Martínez, Raveesh Garg, Michael Pellauer, José L. Abellán, Manuel E. Acacio, and Tushar Krishna. 2023. Flexagon: A Multi-dataflow Sparse-Sparse Matrix Multiplication Accelerator for Efficient DNN Processing. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3* (Vancouver, BC, Canada) *(ASPLOS 2023)*. Association for Computing Machinery, New York, NY, USA, 252–265. https://doi.org/10.1145/3582016.3582069

[34] Quan M. Nguyen and Daniel Sanchez. 2021. Fifer: Practical Acceleration of Irregular Applications on Reconfigurable Architectures. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture* (Virtual Event, Greece) *(MICRO '21)*. Association for Computing Machinery, New York, NY, USA, 1064–1077. https://doi.org/10.1145/3466752.3480048

[35] Tony Nowatzki, Vinay Gangadhar, Newsha Ardalani, and Karthikeyan Sankaralingam. 2017. Stream-dataflow acceleration. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*. 416–429.

[36] Tony Nowatzki, Vinay Gangadhar, Karthikeyan Sankaralingam, and Greg Wright. 2017. Domain Specialization Is Generally Unnecessary for Accelerators. *IEEE Micro* 37, 3 (2017), 40–50. https://doi.org/10.1109/MM.2017.60

[37] Marcelo Orenes-Vera, Esin Tureci, David Wentzlaff, and Margaret Martonosi. 2023. Dalorex: A Data-Local Program Execution and Architecture for Memory-bound Applications. In *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 718–730. https://doi.org/10.1109/hpca56546.2023.10071089

[38] Michael Pellauer, Yakun Sophia Shao, Jason Clemons, Neal Crago, Kartik Hegde, Rangharajan Venkatesan, Stephen W Keckler, Christopher W Fletcher, and Joel Emer. 2019. Buffets: An efficient and composable storage idiom for explicit decoupled data orchestration. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. 137–151.

[39] Raghu Prabhakar, Yaqi Zhang, David Koeplinger, Matt Feldman, Tian Zhao, Stefan Hadjis, Ardavan Pedram, Christos Kozyrakis, and Kunle Olukotun. 2017. Plasticine: A reconfigurable architecture for parallel paterns. *ACM SIGARCH Computer Architecture News* 45, 2 (2017), 389–402.

[40] Raghu Prabhakar, Yaqi Zhang, David Koeplinger, Matt Feldman, Tian Zhao, Stefan Hadjis, Ardavan Pedram, Christos Kozyrakis, and Kunle Olukotun. 2023. RETROSPECTIVE: Plasticine: A Reconfigurable Architecture For Parallel Paterns. (2023). https://bit.ly/isca50_retrospective

[41] Eric Qin, Ananda Samajdar, Hyoukjun Kwon, Vineet Nadella, Sudarshan Srinivasan, Dipankar Das, Bharat Kaul, and Tushar Krishna. 2020. SIGMA: A Sparse and Irregular GEMM Accelerator with Flexible Interconnects for DNN Training. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 58–70. https://doi.org/10.1109/HPCA47549.2020.00015

[42] Alexander Rucker, Matthew Vilim, Tian Zhao, Yaqi Zhang, Raghu Prabhakar, and Kunle Olukotun. 2021. Capstan: A Vector RDA for Sparsity. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture* (Virtual Event, Greece) *(MICRO '21)*. Association for Computing Machinery, New York, NY, USA, 1022–1035. https://doi.org/10.1145/3466752.3480047

[43] Karthikeyan Sankaralingam, Ramadass Nagarajan, Haiming Liu, Changkyu Kim, Jaehyuk Huh, Doug Burger, Stephen W. Keckler, and Charles R. Moore. 2003. Exploiting ILP, TLP, and DLP with the polymorphous TRIPS architecture. *SIGARCH Comput. Archit. News* 31, 2 (May 2003), 422–433. https://doi.org/10.1145/871656.859667

[44] Karthikeyan Sankaralingam, Tony Nowatzki, Vinay Gangadhar, Preyas Shah, Michael Davies, William Galliher, Ziliang Guo, Jitu Khare, Deepak Vijay, Poly Palamuttam, Maghawan Punde, Alex Tan, Vijay Thiruvengadam, Rongyi Wang, and Shunmiao Xu. 2022. The Mozart reuse exposed dataflow processor for AI and beyond: industrial product. In *Proceedings of the 49th Annual International Symposium on Computer Architecture* (New York, New York) *(ISCA '22)*. Association for Computing Machinery, New York, NY, USA, 978–992. https://doi.org/10.1145/3470496.3533040

[45] Nathan Serafin, Souradip Ghosh, Harsh Desai, Nathan Beckmann, and Brandon Lucia. 2023. Pipestitch: An energy-minimal dataflow architecture with lightweight threads. In *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture* (Toronto, ON, Canada) *(MICRO '23)*. Association for Computing Machinery, New York, NY, USA, 1409–1422. https://doi.org/10.1145/3613424.3614283

[46] Dong Kai Wang and Nam Sung Kim. 2021. DiAG: a dataflow-inspired architecture for general-purpose processors. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (Virtual, USA) *(ASPLOS '21)*. Association for Computing Machinery, New York, NY, USA, 93–106. https://doi.org/10.1145/3445814.3446703

[47] Dhananjaya Wijerathne, Zhaoying Li, Manupa Karunaratne, Li-Shiuan Peh, and Tulika Mitra. 2022. Morpher: An open-source integrated compilation and simulation framework for cgra. In *Fifth Workshop on Open-Source EDA Technology (WOSET)*.

[48] Haoran You, Zhanyi Sun, Huihong Shi, Zhongzhi Yu, Yang Zhao, Yongan Zhang, Chaojian Li, Baopu Li, and Yingyan Lin. 2023. ViT-CoD: Vision Transformer Acceleration via Dedicated Algorithm and Accelerator Co-Design . In *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE Computer Society, Los Alamitos, CA, USA, 273–286. https://doi.org/10.1109/HPCA56546.2023.10071027

[49] Jingyang Yuan, Huazuo Gao, Damai Dai, Junyu Luo, Liang Zhao, Zhengyan Zhang, Zhenda Xie, YX Wei, Lean Wang, Zhiping Xiao, et al. 2025. Native Sparse Attention: Hardware-Aligned and Natively Trainable Sparse Attention. *arXiv preprint arXiv:2502.11089* (2025).

[50] Manzil Zaheer, Guru Guruganesh, Kumar Avinava Dubey, Joshua Ainslie, Chris Alberti, Santiago Ontanon, Philip Pham, Anirudh Ravula, Qifan Wang, Li Yang, et al. 2020. Big bird: Transformers for longer sequences. *Advances in neural information processing systems* 33 (2020), 17283–17297.

[51] Guowei Zhang, Nithya Attaluri, Joel S. Emer, and Daniel Sanchez. 2021. 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* (Virtual, USA) *(ASPLOS '21)*. Association for Computing Machinery, New York, NY, USA, 687–701. https://doi.org/10.1145/3445814.3446702

[52] Yaqi Zhang, Alexander Rucker, Matthew Vilim, Raghu Prabhakar, William Hwang, and Kunle Olukotun. 2019. Scalable interconnects for reconfigurable spatial architectures. In *Proceedings of the 46th International Symposium on Computer Architecture*. 615–628.

[53] Yaqi Zhang, Alexander Rucker, Matthew Vilim, Raghu Prabhakar, William Hwang, and Kunle Olukotun. 2019. Scalable interconnects for reconfigurable spatial architectures. In *Proceedings of the 46th International Symposium on Computer Architecture* (Phoenix, Arizona) *(ISCA '19)*. Association for Computing Machinery, New York, NY, USA, 615–628. https://doi.org/10.1145/3307650.3322249

[54] Aojun Zhou, Yukun Ma, Junnan Zhu, Jianbo Liu, Zhijie Zhang, Kun Yuan, Wenxiu Sun, and Hongsheng Li. 2021. Learning n: m fine-grained structured sparse neural networks from scratch. *arXiv preprint arXiv:2102.04010* (2021).

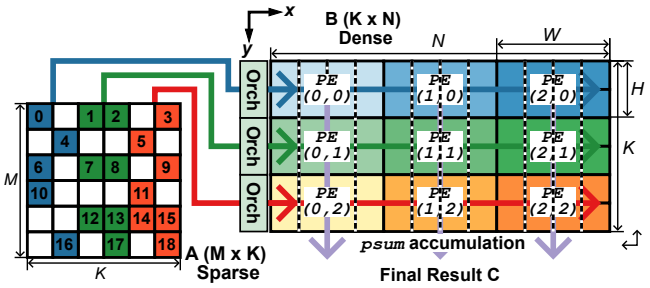


Figure 17. SpMM dataflow

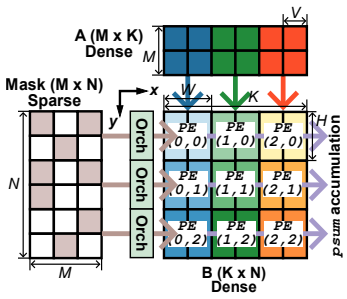


Figure 18. SDDMM dataflow

A Detailed SpMM Mapping

Listing 3 illustrates the high-level dataflow for SpMM. This listing is obtained from transforming the conventional matrix multiplication depicted in Listing 2 by applying loop tiling, reordering, and splitting. The loop is splitted into two parallel tasks: the first, spanning lines 13–22, involves the PEs receiving sparse inputs and executing MAC operations to generate local partial sums, while the second, spanning lines 25–28, handles the asynchronous accumulation of these partial sums, as discussed in Section 4.1.1.

Listing 2. GeMM original code

```
// Memories for inputs and outputs
A[M][K]
B[K][N]
C[M][N]
fn gemm(T* A, T* B, T* C){
 for m in 0..M
  for n in 0..N
   for k in 0..k
    C[m][n] += A[m][k] * B[k][n]
}
```

Listing 3. SpMM Dataflow Pseudo-code (transformed)

```
// See Figure 17 for these hyper-parameters
W,H: width, height of the tile of each PE
M,K,N: Matrix Shapes
X,Y: PE array dimensions
ASSERT W * X = N;
ASSERT Y * H = K;
// Memory Layout
A[M][Y][H]: tiled input sparse matrix A // A[M][K]
B[X][Y][W][H]: tiled input dense matrix B // B[K][N]
C[M][X][W]: output matrix C // C[M][N]
psum[M][X][Y][w]; // intermediate psums produced by PE(X, Y)

fn spmatmul(T* A, T* B, T* C){
 // local psum computation
 for m in 0..M // sequential execution (temporal)
  for h in 0..H // sequential execution (temporal)
   parallel_for x in 0..X // parallel execution (spatial)
    if (A[m][y][h] != 0) {// executed by orchestrators
     parallel_for y in 0..Y // parallel execution (spatial)
     // scalar-vector multiplication and accumulation
      _sv_mac(psum[m][x][y], A[m][y][h], B[x][y][h]);
    } // End if non-zero

 // accumulation of partial sums (asynchronous, non-associative)
 for (a = 0: a < Y; a++) //
  for (b = 0; b < X; b++) //
   for (m = 0; m < M; m++) //
    _vv_acc(psum[m][a][b], C[m][a]);
}
```

B Detailed SDDMM Mapping

Listing 4 illustrates the SDDMM dataflow. In each cycle, a vector from matrix A with width V (corresponding to the vector lane size) is streamed into the first PE from the top. Simultaneously, a bitmask M is provided to the orchestrator to select the column of B that will be dot-multiplied with the incoming A vector. Each PE then executes element-wise MAC operations between the A vector and the corresponding locally stored B vector. Each PE processes H columns of B; these columns are partitioned into $\frac{H}{V}$ cycles of vectored MAC operations. This process reduces the accumulation dimension from H to V, yielding a partial sum vector of width V. These partial sums are propagated from left to right and are accumulated in a V dimension vector. Finally, the reduction of the V-dimensional partial sum to a single scalar is performed by the last column of PEs, just before the result is forwarded to the memory controllers.

Listing 4. SDDMM Dataflow Pseudo-code

```
// See Figure 18 for these hyper-parameters
W, H: width, height of the tile of each PE
M,K,N: Matrix Shapes
X,Y: PE array dimensions
V: Vector lane width of each PE
ASSERT W * X * V = K
ASSERT Y * H = N;
// Memory Layout
A[M][W][X][V]: tiled input sparse matrix A // A[M][K]
B[X][Y][H][W][V]: tiled input dense matrix B // B[K][N]
C[M][Y][H]: output matrix C // C[M][N]
M[M][Y][H]: the mask of the result matrix // M[M][N]
psum[X][Y][V]
psum_reduced[Y][V]
psum_final[Y]

fn sddmm(T* A, T* B, T* C, T* M){
 for m in 0..M // sequential execution (temporal)
  for h in 0..H // sequential execution (temporal)
   for w in 0..W // sequential execution (temporal)
    parallel_for x in 0..X // parallel execution (spatial)
     parallel_for y in 0..Y // parallel execution (spatial)
      if (M[m][y][h] != 0){
       _vv_mac(psum[x][y], A[m][w][x], B[x][y][h][w])
      }
 for y in 0..Y
  for x in 0..X
   // reduce psum from shape (x,y,v) to (y,v)
   _vv_acc(psum_reduced[y],psum[x][y])
 for y in 0..Y
  for v in 0..V
   // reduce to the final psum
   psum_final[y] = _sum(psum_reduced[y])
}
```

C FSM behavioural insights

Figure 19, 20, and 21 show three representative snapshots of the SpMM kernel execution. For illustration purpose, we use the same dense matrix B for mapping as in Figure 17, where each PE stores only two rows of B. For clarity, the figures display only the orchestrator's internal buffer-management state and the first PE of each row, focusing on the top two rows of the PE array to illustrate their interaction. To simplify, we consider a scratchpad of size 4.

Case 1 (Normal Operation). When a new non-zero element arrives at the orchestrator's input and no message is received from a neighboring orchestrator, the orchestrator issues a MAC operation for its row of PEs. It calculates the address in the local memory (i.e., a slice of B) based on the column index of the incoming element from A, ensuring that each PE fetches the correct data from its local memory. For instance, if the input metadata in the first row indicates a non-zero element from A at row 4, column 1 (case ①, A[4][1]), the local memory address that the PE needs to load from the corresponding B matrix is computed as (1 modulo 2), since two rows of B are mapped to each row of PE.

Case 2 (Row Completion and Psum Forwarding). Once a row finishes processing, indicated by a row-end token, space in the scratchpad must be freed for a new row. According to our FIFO buffer management policy, the oldest buffered partial sum (psum) is flushed to the downstream PE. The state meta registers record the row ID (RID) of the oldest element and its corresponding offset in the scratchpad (to implement a circular FIFO).

The orchestrator also informs the downstream orchestrator about the row index of the forwarded psum (e.g., psum(1) for the psum of row index 1[2]). Upon receiving this message, the downstream orchestrator checks whether the row index of the incoming psum falls within its current range of responsibility (based on the index of the newest buffered row and the buffer length). If it does (as in case 2), the downstream row stops its local psum computation and accumulates the incoming psum into its scratchpad.

[2]psum(1)represents the inter-orchestrator message, while psum[1] denotes the actual value.

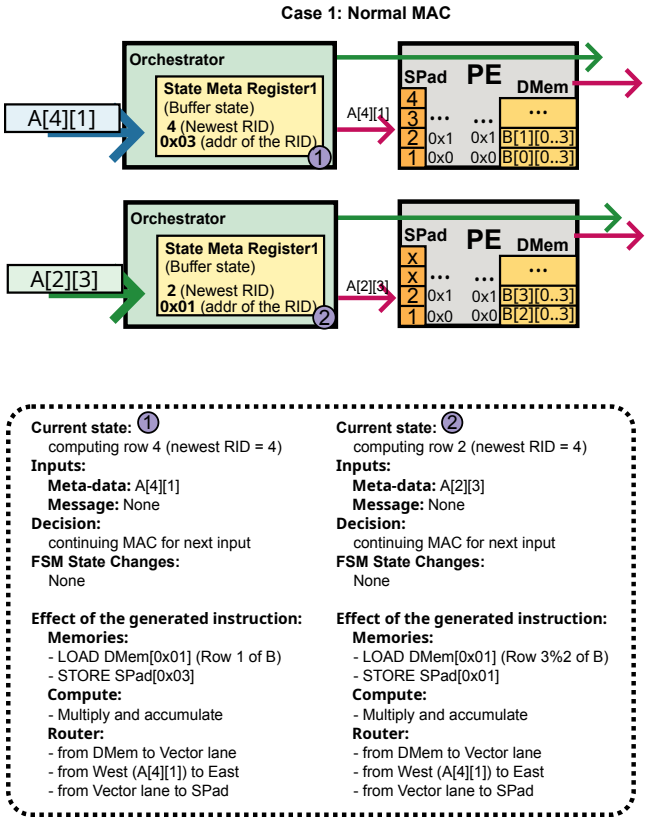


Figure 19. SpMM detailed execution: case 1

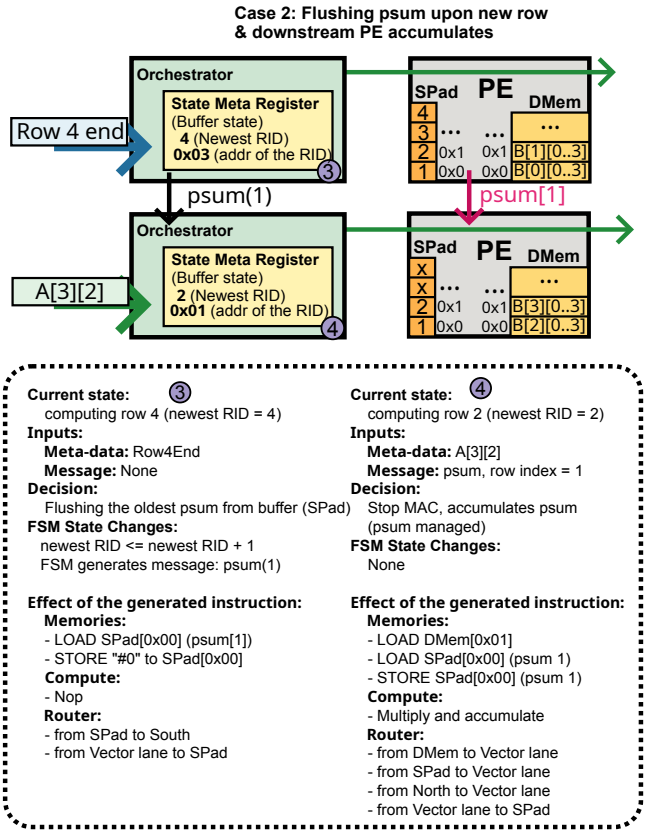


Figure 20. SpMM detailed execution: case 2

Case 3 (Early psum Arrival and Bypass). If the downstream orchestrator receives a psum whose row index falls outside its current range—in this case, receiving psum 5 while the newest row being computed is row 2—it bypasses the received psum without interrupting its local computation. This situation indicates a workload imbalance, as the downstream orchestrator is "too late," potentially due to an excessive number of sparse elements in its previous row. The bypass is achieved using the router by directly forwarding the psum from north to south without interrupting the execution pipeline. The downstream orchestrator also notifies its next downstream orchestrator about the bypassed psum.

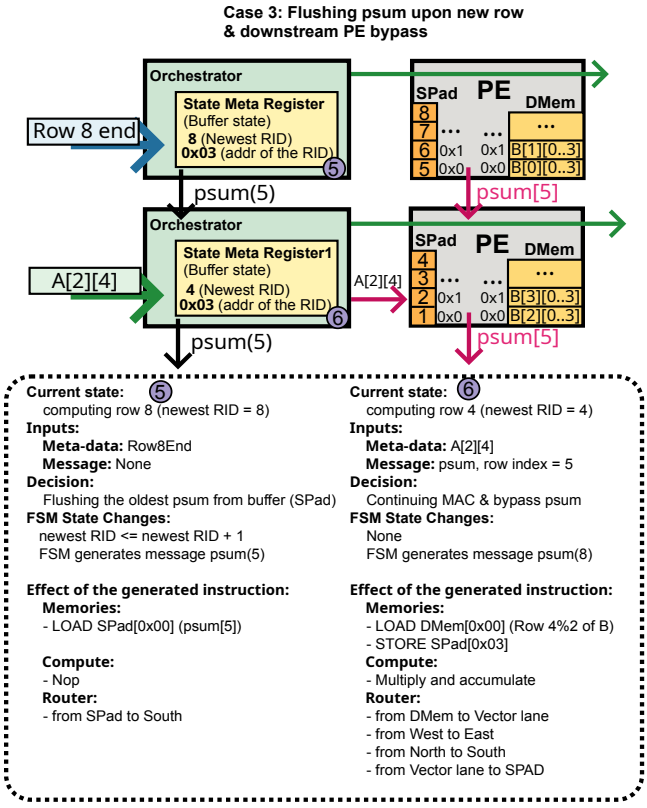


Figure 21. SpMM detailed execution: case 3

D Using Canon for Spatial Execution

Canon inherently can support a fully static place-and-route style spatial mapping, in which each PE executes the same instruction over time. This mapping is typically associated with entirely spatially reconfigurable architectures where a kernel's dataflow graph can be entirely mimicked on the fabric. Figure 22 illustrates how this is accomplished. During the configuration phase, the orchestrator preloads instructions into the PE array without immediate execution—the results are discarded. For instance, fully configuring a 4-column PE array requires 12 cycles (4 columns × 3 cycles). During the execution phase, PEs can be set to a "hold" state to prevent instructions from propagating further through the pipeline to subsequent PEs. This effectively means stopping the *staggered instruction issue*. This mechanism allows the PEs to execute the preloaded instruction from the configuration phase, with the orchestrators maintaining the hold signal to ensure that the PEs remain in this state.

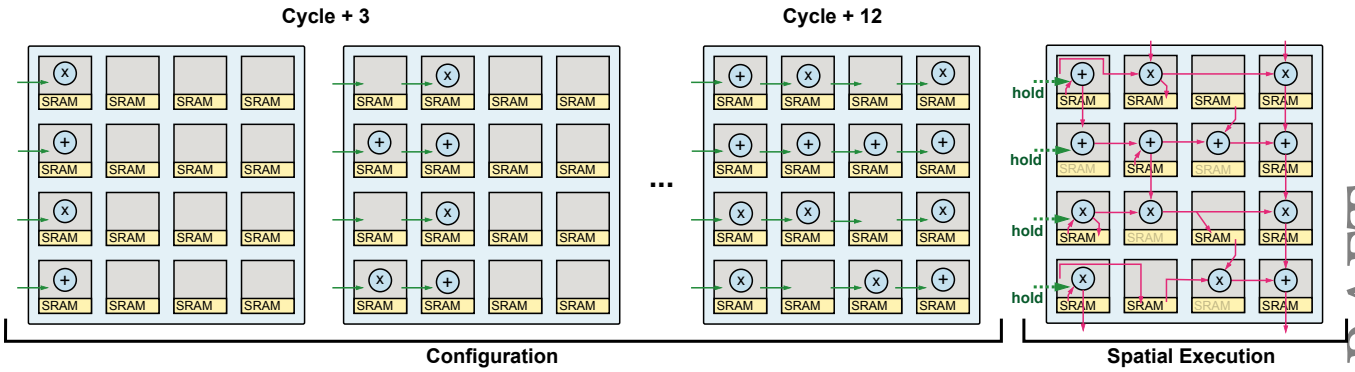


Figure 22. Using *Canon* for spatial dataflow execution