# On the Design of a Software-Controlled Memory System for the Atalla Ax01 Artificial Intelligence Accelerator

**Authors**

| | | | |
|---|---|---|---|
| Akshath Raghav Ravikiran | Julio Hernandez | Haejune Kwon | Rafael Monteiro Martins Pinheiro |
| *araviki@purdue.edu* | *herna628@purdue.edu* | *kwon196@purdue.edu* | *rmontei@purdue.edu* |
| ECE 69600 | VIP 47921 | VIP 47920 | VIP 47920 |

***Elmore School of Electrical And Computer Engineering***
***Purdue University, West Lafayette, IN***

**Submitted To:**
Fan Jing Hoon
Malcolm McClymont
Sooraj Chetput
Timothy Hein
Rishikesh Bhatina

# Table of Contents

## Abstract

Modern AI Accelerators, like the Atalla Ax01 core, depend on exploiting predictable, high-bandwidth data movement between on-chip memory and compute modules. The Machine Learning (ML) workloads that these specialized chips target inherently expose deterministic access patterns – including tiled matrix multiplication, toeplitz-based convolution, tensor transposes and vector operations. The memory primitive in such architectures are arrays, often called vectors, as opposed to the scalar-optimized general-purpose chips. Additionally, these chips implement memory hierarchies that rely on conventional cache designs, accommodating a wide variety of workloads. These mechanisms are defined by tag overheads, unpredictable latencies, and hardware-based prefetchers that are optimized for adapting to different workload characteristics at runtime.

In order to study alternate designs that exploit the aforementioned data regularity, Atalla Ax01's "Scratchpad" architecture implements a parameterizable software-controlled memory subsystem that combines (1) a tile-descriptor-based DMA engine (2) SRAM-banking strategies evaluated on different technology nodes, and (3) four multi-stage interconnect topologies that re-arrange vectors on-the-fly. The architecture provides for swizzle-based movement, avoiding bank-conflicts for wide vector-reads and transpose-friendly micro-op scheduling.

A core component of the explored design space were the interconnect micro-architectures - Benes, Batcher-Banyan, CLOS - simply referred to as crossbars. All designs were synthesized on the MIT-LL 90nm CMOS process node, and optimized for area, power and clock frequency. A Python-based emulator was built to confirm the viability of the design for the targeted behaviour, and simulated against every possible data-access pattern. The micro-architecture was implemented

and modelled at a gate-level using SystemVerilog - within the industry-standard QuestaSim - achieving code coverage of 90%+.

The work accomplished by this team has laid the foundation for a hardware-software co-designed kernel library that will provide optimized implementations of core vector and matrix operations. In parallel, a custom compiler is being developed to lower this code into instruction bundles, enabling parallel execution of μ-ops to use the asynchronous data-movement features of the Scratchpad. The Atalla Ax01 accelerator will feature a 4-wide tainted-VLIW scheduler, split-transaction AXI-bus and dual-channel DDR4 controller to complement the Scratchpad design.

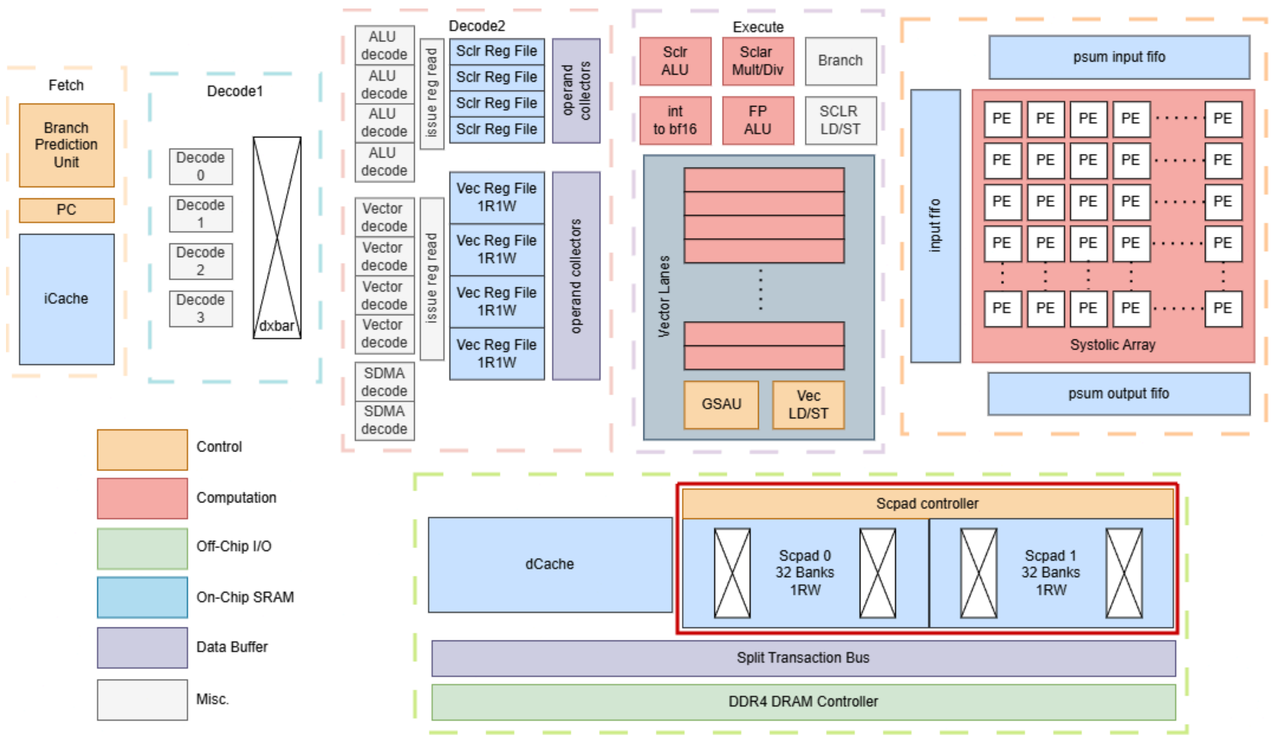


*Figure A: Top Level Atalla Ax01 Architecture*

# Chapter 1: Background

## *Section 1.1: Machine Learning Workloads*

The solutions to many real-world problems cannot be addressed with explicitly programmed rules or heuristics. Natural language understanding, speech recognition and image classification require alternate approaches to manual programming — they require learning from data and patterns. These applications demand trainable "models" that are flexible enough to approximate complex functions, all in the effort to mimic the human brain's capabilities.

Machine Learning (ML) is the formal field that defines the mathematical processes needed to synthetically learn from the world. Generally, these processes involve repeatedly updating a large set of floating-point values, called "weights", that help align itself towards approximating closest to the expected answer. Modern ML models utilize multiple such processes, incorporating them into "layers" for hierarchically abstracting upwards. However, from a computing perspective, these layers repeatedly utilize a fixed set of representative computer programs – optimized to run on different computing paradigms – called "kernels". The dominating kernels can be identified as General Matrix Multiplications (GEMMs) and Convolutions (CONVs).

GEMMs are the linear algebra operation $C = A \times B$, with $A$, $B$ and $C$ matrices. They are inherently computationally dense processes and highly parallelizable, making them ideal for hardware acceleration. In Section [1.2], we explore a common strategy used to squeeze out even more performance from these kernels by taking advantage of the "locality" principle in general-purpose chips. CONVs, on the other hand, slide small filters called feature maps across matrices and compute a weighted sum at each spatial location. CONVs are generally transformed into GEMMs to take advantage of general-purpose memory hierarchies. Methods like Im2Col and Im2Row transform intermediate operands to maximize re-use and simplify instruction scheduling[1]. Thus, it is possible to efficiently compute both GEMMs and CONVs in the same piece of hardware, given this transformation.

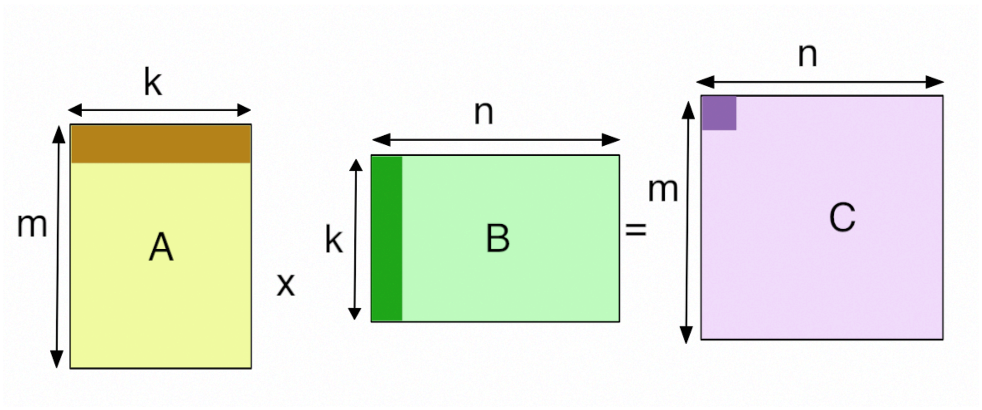


*Figure 1.1.1: GEMM Visualization*[2]

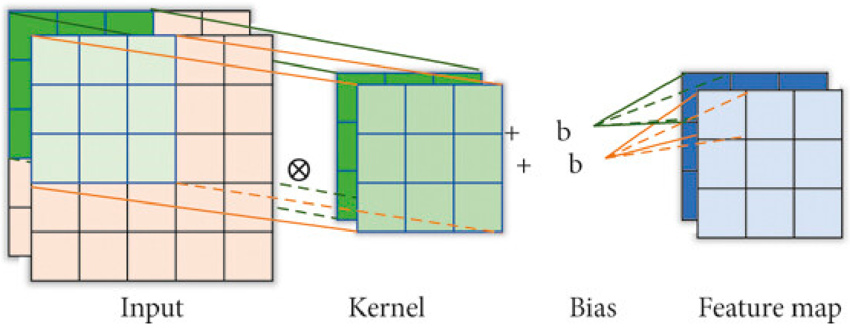


*Figure 1.1.2: CONV Visualization*[3]

Memory locality is a concept that states that a computer program has a tendency to access the same set of memory locations repeatedly over a contiguous interval of time. In ML workloads, memory locality arises because the core algorithms have a tendency of accessing weights and activations multiple times for the same set of outputs, and also will access the matrices close to the ones that have been accessed recently.

## *Section 1.2: Custom Matrix Multiplication Hardware*

As the size of operand matrices increases, multiplying individual matrix values sequentially using naive pipelines leads to drastically worsened program completion time. Additionally, at scale, each matrix cannot fit inside the on-chip memory structures and a lack of cache-aware programming leads to memory bandwidth under-utilization. To deal with this mismatch, we make use of tiling. Tiling breaks down big input matrices/feature maps into smaller patches, called tiles, that fit within on-chip memory and can be fed into the compute pipeline without incurring avoidable off-chip memory accesses. Moreover, this concept exposes the opportunity for data re-use, but requires thoughtful kernel engineering for maximizing it.

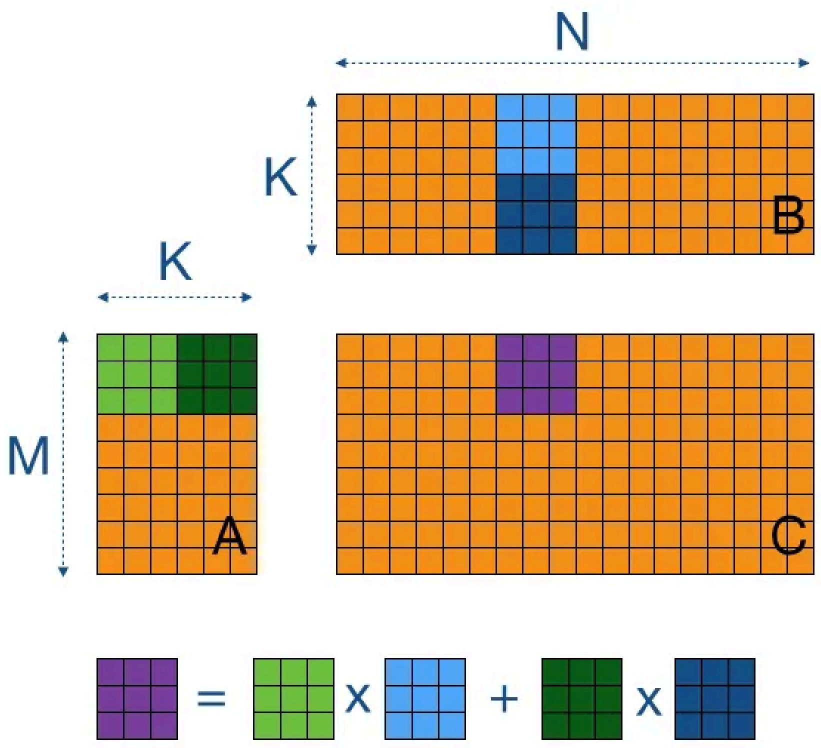


*Figure 1.2.1: Tiling over a GEMM Operation*[4]

Upon a closer look, GEMMs and CONVs can be realized as very regular matrix operations that can be easily parallelized. This allows for a structured compute fabric to be designed for exploiting regularity – the Systolic Array. A Systolic Array is a grid of processing elements (PEs) arranged in rows and columns. Each PE performs a multiplication of two incoming values, accumulates the value, and then passes the intermediate results to neighboring PEs. This pipelined operation is called a Multiply-Accumulate (MAC). The data is reused across the array, reducing the number of values fetched from memory. Instead of performing multiplications sequentially, we can now improve computation throughput by streaming "slices" of the matrices into the Systolic Array. Moreover, such a design compliments the tiling paradigm, further maximizing re-use across the global matrix-matrix multiplication schedule. It is for this reason that Systolic Arrays are found in almost all custom AI Accelerators.

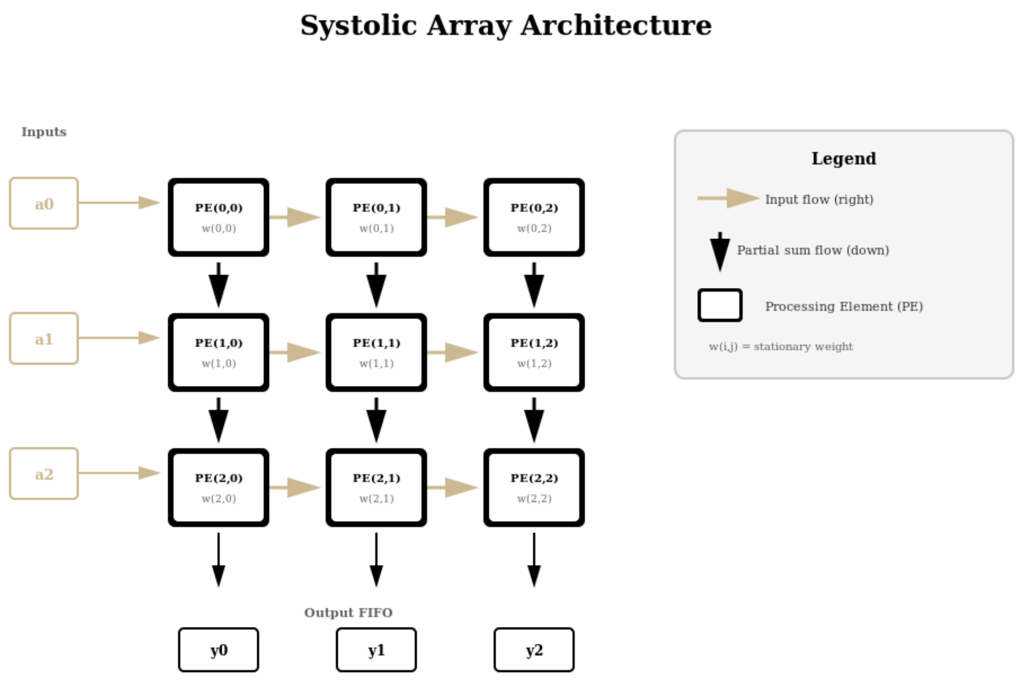


*Figure 1.2.2: 3x3 Weight Stationary Systolic Array Architecture*[5]

With the dominance of Systolic Arrays, we want to be able to utilize the same hardware to speedup CONVs. Consider the GEMM operation $C = A \times B$. To convert a CONV into a GEMM, we would have to identify the transformations: (1) $f\colon A \to A'$, (2) $g\colon B \to B'$, and (3) $h\colon C' \to C$ , such that $C' = A' \times B'$ and $C = A * B$ holds true. In doing so, we can perform a GEMM with $A'$ and $B'$ to get $C'$ , a transformed yet invertible result of the original CONV operation. The Im2Col algorithm, previewed in Section [1.1], does just this.

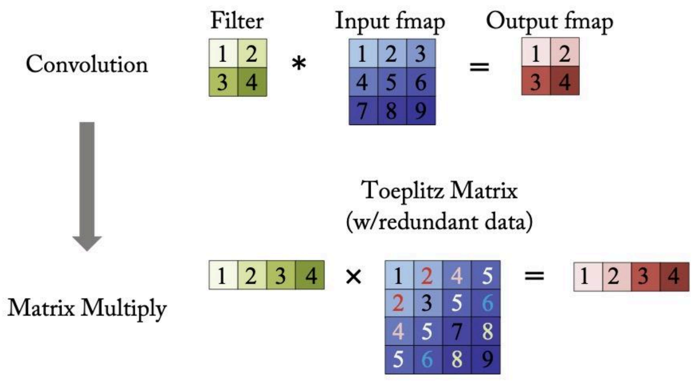


*Figure 1.2.3: Toeplitz Generation; Im2Col Idea.* [6]

The Im2Col algorithm utilizes the Toeplitz transform, a function that unfolds input patches into matrix rows $(f)$ and rearranges convolution feature maps into another  matrix $(g)$. The Toeplitz matrix generated by this transformation encodes all sliding-window regions of the input feature map. Once the output matrix is produced using the core GEMM hardware, it can be reshaped back into the original feature-map layout $(h)$ for the CONV process. From our exploratory readings, we identify that Google's TPUs and Nvidia's Tensor Cores implicitly generate these Toeplitz matrices instead of creating these redundant data duplications in the DRAM [1].

## *Section 1.3: DMA Engines*

Direct Memory Access (DMA) engines are dedicated hardware components that are responsible for movement of data directly from main memory and on-chip caches without involving the processor in every single operation. A DMA operation can be programmed to read an entire tile from matrix $B$, store it in local memory and set up the next operation while the computation is still engaged with operations on matrix $A$. As such, these engines de-couple the memory and compute datapaths within a single instruction stream. Specialized DMA instructions and intrinsics need to be defined for proper operation of such structures. While caches operate on a result-driven mode, resulting in unpredictability behaviour that is hidden by the hardware, DMA operations are predictably exposed to the controlling software.

## *Section 1.4: Crossbars and the Need for Swizzling*

The Systolic Array assumes that data will be provided in a specific pattern and manner, which is not taken into account by either tiling or DMA engines. Unfortunately, as it often happens, data accesses to local memory do not align with a specific pattern. As a result, there are inefficiencies due to accesses that, if not addressed, results in stalls, inefficient memory usage and unused computation resources.

Caching structures that store the operand tiles are usually implemented with multiple independent banks so that more than one value can be read or written in parallel. However, if multiple simultaneous accesses target the same bank, it leads to a bank conflict and consequent serialization of accesses. Within the Systolic Array, in which hundreds of MAC operations occur simultaneously, a small number of conflicts can significantly impact performance.

To address these issues, swizzling –  a deterministic data remapping – is employed to put the data into a form that is most computational-friendly before it proceeds into the compute fabric. Swizzling re-arranges the mapping of data to banks so that simultaneous accesses are distributed in such fashion that there are no bank conflicts, enabling the Systolic Array to process at maximum throughput. To achieve swizzling, there would have to be a mechanism on the hardware side that could dynamically rearrange data with the objective of routing a given logical unit to a different physical address based on the swizzling pattern. The basic component on the hardware side that would make this feasible is called an interconnect, or crossbar.

Crossbars are any system that routes input signals to any of the outputs, allowing for rearrangement of data on-the-fly. These designs are essential for network systems, with the origin from telecommunication exchanges, where flexible signal rearrangement was required on a large scale. With fast development of digital systems and their complexity, such as GPU and memory systems, the need for efficient crossbars is increasing much more than before. The strong effort of engineers to reduce latency with minimal area and power has led to various multistage interconnection networks, which consist of multiple stages of smaller and simpler switches, reducing hardware cost while performing permutations with low latency.

# Chapter 2: Introduction

## *Section 2.1: Trends in Machine Learning Workloads*

Modern ML workloads, especially since the advent of large reasoning models, have grown in memory capacity and bandwidth requirements, pushing architectures to target higher peak Floating-Point Operations (FLOPs) under relatively tighter power budgets. The Memory Wall, described as the difference in scaling over time between processor speed and memory bandwidth, also explains how even the highest throughput-capable memory cards remain the bottleneck for model inference in naive pipelines. This underpins the importance for capable on-chip cache-like structures to help adjacent compute units and scheduling kernels to optimize for higher achievable throughput of AI accelerators.

As discussed in Section [1.1], the dominant ML layers share a fixed set of core kernels – tiled matrix multiplication, toeplitz-based convolution, tensor transposes and vector operations – that exhibit highly regular, compile-time-aware memory access patterns. These kernels abstract away the handling of lower-level constructs provided by the specialized compilers, and are optimized to reach maximum empirical performance. The nature of tiling simplifies the scheduling of these kernels, and constrains the number of these access patterns. Moreover, tiling across multiple-levels for these workloads retains a base set of repeating patterns that can be memoized and ingrained into the hardware. Unlike runtime-data-dependent workloads, ML workloads allow for a cleaner hardware-software co-design where the memory coordination can be optimized for a simpler set of possible instruction sequences and parallelism can be exploited at the kernel level instead.

## *Section 2.2: Need for Specialized Memory Systems*

However, taking advantage of this regularity requires careful optimizations downstream, especially in the memory-compute bridge. Matrix-based arithmetic units, as discussed in Section [1.2], are deeply-pipelined to exploit this feature. Consequently, data movements need to be refined for supplying those units at a sustained rate and to optimize overall throughput. This "feeding" also needs to ensure high utilization percentages across all such units over time, which means these memory structures need to directly expose parallelism in the hardware – either by asynchrony or by banking. A few missed cycles of feeding can propagate multi-cycle bubbles throughout these fabrics, preventing them from reaching even half of their theoretical performance. These strict temporal constraints impose certain demands from the memory system that are vastly different from the expectations of a general-purpose cache hierarchy.

Traditional caches attempt to exploit both temporal and spatial locality of data accesses in the hardware. It is indexed with memory addresses, and uses tag comparisons to confirm storage location of a piece of data. Caches, in most modern out-of-order and superscalar processors, implement prefetching and multi-level replacement policies to identify seemingly unnoticeable patterns in sequential data streams. However, these methods incur considerable silicon area and would provide no benefit to the targeted workloads. Moreover, the alignment of vector data, in such caches, cannot be guaranteed to satisfy conflict-free reads from a single frame; state-of-the-art software stacks for high-performance CPUs provide special intrinsics as a workaround, but require considerable changes to existing codebases for noticeable speedup. The unpredictable and runtime-based behaviours of conventional caches remove them from consideration for AI accelerator architectures. Moreover, the use of tag bits (which usually take up ~6% of the cache's real capacity) can be avoided if the location of every word is known at compile-time.

Instead, recent designs focus on using software-managed caches, called Scratchpads, that are implemented as multi-banked SRAM arrays controlled by specialized compilers at runtime. This necessitates specialized low-level primitives to control the banks, consequently enabling an optimal (OPT) replacement schedule to handle eviction and aging. These compilers are able to exploit fine-grained scheduling optimizations and ensure wide vector accesses are aligned and agnostic of DRAM memory addresses. The inclusion of Scratchpad-like designs leads to challenges that need to be addressed in both hardware and software.

## *Section 2.3: Need for Multi-Stage Interconnects*

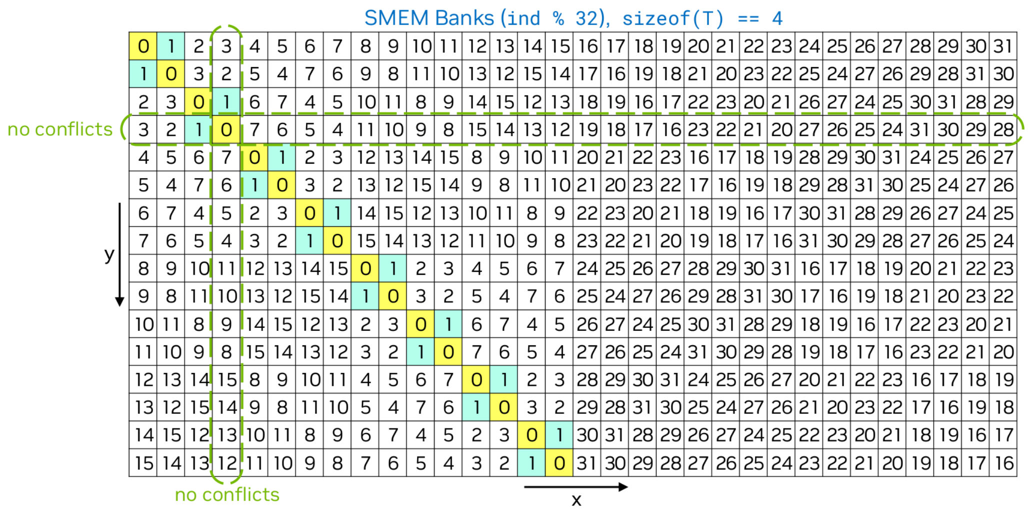
Bank conflicts in traditional caches take place when multiple data blocks exist in the same SRAM bank. Computer architectures optimized for scalar workloads can work around this problem using dynamic scheduling schemes and exploiting MLP (with MSHRs) to improve instruction commit rate. However, Superscalar and vector architectures require wide (or parallel) accesses. To perform a wide fetch of data in the presence of bank conflicts would require serially accessing the same bank across multiple cycles, prolonging a program's completion time.

Solving bank conflicts in ML workloads requires us to understand the nature of possible accesses. Sliding windows used in CONVs, if implemented naively with a hardware-controlled cache, will deterministically incur high bank conflicts when creating a Toeplitz patch. Similarly, matrix transpose operations will attempt to read multiple values that belong to the same SRAM bank in a single cycle. A common technique for mitigating these collisions, also utilized to reduce aliasing in other hardware mechanisms, is to use an XOR Swizzle. This invertible function, defined below in Figure [2.3.1], is used to guarantee that any logical row or column of a tile can be reordered to ensure that values belonging to the same row or column can be accessed in a single wide access. The XOR gate, symbolized as ⊕, forms of the core of this mechanism. Below, "r" and "c" denote a binary index value, while NUM_BANKS is the number of times a cache is banked. A visualization of the XOR Swizzle can be understood using Figure [2.3.2].

$$bank(r, c) = (c \oplus (\mathrm{r\ mod\ NUM\_BANKS}))\ \mathrm{mod\ NUM\_BANKS}$$

*Figure 2.3.1: XOR Swizzle*

These formulas can be realized with simple XOR gates, and find use in convolution and transpose µ-op sequences found in GPUs. Following Amdahl's law, which helps contextualize architectural trade-offs based on the commonality in representative workloads, many custom accelerators implement swizzling in some manner because of the benefit they pose to ML kernels. For example, AMD Instinct GPUs use Morton Ordering [7] – a kind of swizzle function – and expose it via their HIP programming model [8].



| | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 | 29 | 30 | 31 |
| 1 | 0 | 3 | 2 | 5 | 4 | 7 | 6 | 9 | 8 | 11 | 10 | 13 | 12 | 15 | 14 | 17 | 16 | 19 | 18 | 21 | 20 | 23 | 22 | 25 | 24 | 27 | 26 | 29 | 28 | 31 | 30 |
| 2 | 3 | 0 | 1 | 6 | 7 | 4 | 5 | 10 | 11 | 8 | 9 | 14 | 15 | 12 | 13 | 18 | 19 | 16 | 17 | 22 | 23 | 20 | 21 | 26 | 27 | 24 | 25 | 30 | 31 | 28 | 29 |
| 3 | 2 | 1 | 0 | 7 | 6 | 5 | 4 | 11 | 10 | 9 | 8 | 15 | 14 | 13 | 12 | 19 | 18 | 17 | 16 | 23 | 22 | 21 | 20 | 27 | 26 | 25 | 24 | 31 | 30 | 29 | 28 |
| 4 | 5 | 6 | 7 | 0 | 1 | 2 | 3 | 12 | 13 | 14 | 15 | 8 | 9 | 10 | 11 | 20 | 21 | 22 | 23 | 16 | 17 | 18 | 19 | 28 | 29 | 30 | 31 | 24 | 25 | 26 | 27 |
| 5 | 4 | 7 | 6 | 1 | 0 | 3 | 2 | 13 | 12 | 15 | 14 | 9 | 8 | 11 | 10 | 21 | 20 | 23 | 22 | 17 | 16 | 19 | 18 | 29 | 28 | 31 | 30 | 25 | 24 | 27 | 26 |
| 6 | 7 | 4 | 5 | 2 | 3 | 0 | 1 | 14 | 15 | 12 | 13 | 10 | 11 | 8 | 9 | 22 | 23 | 20 | 21 | 18 | 19 | 16 | 17 | 30 | 31 | 28 | 29 | 26 | 27 | 24 | 25 |
| 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 |
| 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 24 | 25 | 26 | 27 | 28 | 29 | 30 | 31 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 |
| 9 | 8 | 11 | 10 | 13 | 12 | 15 | 14 | 1 | 0 | 3 | 2 | 5 | 4 | 7 | 6 | 25 | 24 | 27 | 26 | 29 | 28 | 31 | 30 | 17 | 16 | 19 | 18 | 21 | 20 | 23 | 22 |
| 10 | 11 | 8 | 9 | 14 | 15 | 12 | 13 | 2 | 3 | 0 | 1 | 6 | 7 | 4 | 5 | 26 | 27 | 24 | 25 | 30 | 31 | 28 | 29 | 18 | 19 | 16 | 17 | 22 | 23 | 20 | 21 |
| 11 | 10 | 9 | 8 | 15 | 14 | 13 | 12 | 3 | 2 | 1 | 0 | 7 | 6 | 5 | 4 | 27 | 26 | 25 | 24 | 31 | 30 | 29 | 28 | 19 | 18 | 17 | 16 | 23 | 22 | 21 | 20 |
| 12 | 13 | 14 | 15 | 8 | 9 | 10 | 11 | 4 | 5 | 6 | 7 | 0 | 1 | 2 | 3 | 28 | 29 | 30 | 31 | 24 | 25 | 26 | 27 | 20 | 21 | 22 | 23 | 16 | 17 | 18 | 19 |
| 13 | 12 | 15 | 14 | 9 | 8 | 11 | 10 | 5 | 4 | 7 | 6 | 1 | 0 | 3 | 2 | 29 | 28 | 31 | 30 | 25 | 24 | 27 | 26 | 21 | 20 | 23 | 22 | 17 | 16 | 19 | 18 |
| 14 | 15 | 12 | 13 | 10 | 11 | 8 | 9 | 6 | 7 | 4 | 5 | 2 | 3 | 0 | 1 | 30 | 31 | 28 | 29 | 26 | 27 | 24 | 25 | 22 | 23 | 20 | 21 | 18 | 19 | 16 | 17 |
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 |

*Figure 2.3.2: Swizzling in Shared Memory (SMEM)* [9]

Any tiling strategy chosen by ML kernel developers will produce varying issuing schedules based on parameters like tile-size, tensor-shape, rank-traversal-order, stride and dilation. These factors decide how efficiently work can be scheduled into the compute units, and at what rate. Initiation Interval (II) is a value that defines the cycles between two successive requests consumed by a module. Streaming workloads can be optimized to ensure II = 1, which means that data can be fed into the compute units every cycle. On the flip side, this also means that a single cycle where the producer is stalled could cascade into many cycle bubbles in the consumer. This ties back to the discussion in Section [2.2] regarding the need to keep Systolic Arrays and Vector Cores fed constantly. Many AI accelerator designs are based on this core idea: GEMM, CONVs and Vector Operations can be streamed for tile-based constructs. This means that they need to ensure that swizzling of data can happen, in either a combinational or fully-pipelined manner,  after accessing the data from cache and before entering the compute units.

For this reason, many architectures, even those targeting non-AI workloads, implement some form of an on-chip interconnect to ensure that all producing slots can write to all consuming slots. While the word "interconnects" varies in implementation across different domains, they are commonly referred to as "crossbars" in chip design. To adapt to the tight temporal constraints, a single-cycle all-to-all crossbar would be the simplest topology. However, these designs demand a large silicon area, incur heavy power draws once synthesized, and drastically limit maximum clock frequency. We believe Nvidia moved away from using crossbars in their GPUs for this reason, instead focusing on statically-mapped compiler-managed datapaths between their operand collectors and register files. To repeat, this is owing to the large number of crosspoints between wires connecting ports, as well as the large N:1 muxes that route the values to the correct destination. Thus, alternative designs were developed, specifically employed in the networking and high-performance-computing (HPC) domains, to balance the trade-offs between area, power, delay, flexibility, and scalability. These topologies will be discussed in Section [3.6] and are well studied to understand their tradeoffs in data centers. However, their synergy with ML-focused datapaths remain minimally explored.

## *Section 2.4: Need for DMA Support*

CPUs are designed to provide a workload agnostic way to load data from DRAM into on-chip memory using a one-dimensional addressing scheme. While programmers get this view, the physical layout of RAM structures are hierarchical but give the illusion of addresses being mapped linearly. Compilers optimize their binaries with this concept to improve cache-accessing bandwidth through alignment and padding. By doing so, the hardware prefetching and load-store speculation mechanisms can identify patterns and pre-emptively load data onto the chip, sacrificing some bandwidth. But, AI Accelerators ignore these complications completely. The workload can be abstracted hierarchically, from the programmer's standpoint, such that every scheduling decision to load tiles from memory would lead to a repeated set of accesses whose pattern is known at compile time. In other words, the compiler does not need to emit a "load" instruction for every data word that it would like to place into the cache. A special de-coupled circuit could be defined to traverse through the pattern without manual per-cycle control.

To exploit this, accelerator architectures and software stacks implement direct-memory-access (DMA) engines in the hardware to load all pieces of data within a tile onto the chip. This concept was introduced in Nvidia's Hopper architecture [10], where they formalize it using a hardware module called the Tensor Memory Accelerator (TMA). Nvidia's complimentary CuTe library allows programmers to define "events" that load tiles, or sets of tiles, by attaching some metadata regarding on-chip address, off-chip address, swizzling format and other software-scheduling-focused knobs. This abstraction has been used in some of the fastest kernels on the market, which asynchronously load data through a de-coupled TMA while other compute units consume data parallely. The

FlashAttention4 kernel uses this idea, hidden under the innovation of warp-specialization, to overlap and de-couple work done by parts of a highly-parallel hardware. However, the details of the TMA and the supporting NVCC compiler constructs are not public information, and are missing open-source implementations for community evaluation.

## *Section 2.5: Scratchpad Motivation*

This lack of open-source implementations has created a gap in understanding tradeoffs when exploring the design space of creating accelerators. Industry giants like Nvidia and AMD do not open source their micro-architecture designs, but document support for swizzling capabilities in their CUTLASS and ROCm libraries [7]. The choice of interconnect and SRAM banking strategy would greatly influence the compiler architecture and the programming model exposed for end-users of any accelerator design. This project aims to fill this gap by implementing all the components of a memory subsystem designed around ML workload regularity, and parameterized to allow for plug-and-play functionality when configuring architectural "schmoos". The aim is to ensure that the end-users of Atalla Ax01's emulation platform can sweep the hyperparameter space and evaluate data-flow specific designs for evolving use-cases. Additionally, all implementations are open-sourced, along with the physical design flows set up for MIT-LL's 90nm CMOS process and an emulator for verifying functional correctness. Future work entails further development of a programming model and kernel library, with automations standardized to map the RTL design onto an FPGA.

# Chapter 3: Architecture

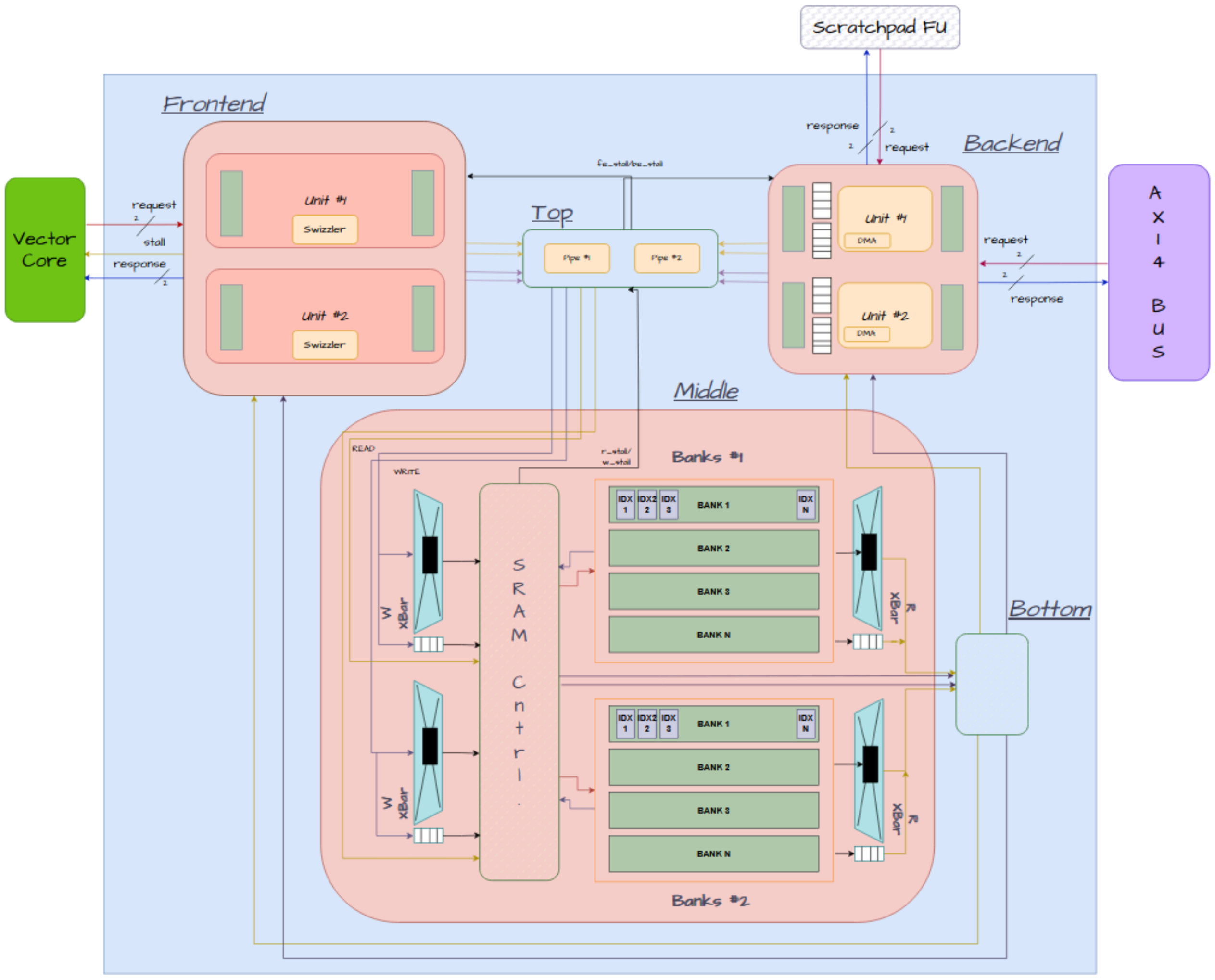


*Figure 3.0: Scratchpad Architecture*

## *Section 3.1: Top-Level Overview*

The Scratchpad subsystem is controlled by the Scheduler Core through the “Scratchpad FU” interface shown in Figure [3.0]. The 4-wide VLIW Scheduler issues tile-level load and store operations to the Scratchpad, stalling bundle flow until sequential completion. This forces the compiler to re-order instructions for the worst-case latency. On the datapath, the Scratchpad interfaces directly with the Vector Core, which buffers and offloads data to the Systolic Array. With the help of a split-transaction AXI Bus, it is able to work with the DDR4 DRAM controller to transfer tiles to and from external memory. With such responsibilities, the Scratchpad can be viewed as the central on-chip location where tiles are buffered, consumed and overwritten by the compute units.

The micro-architectural implementation of this design is based on a fixed Maximum Tile Size, which is known to the compiler. In defining this value, the compiler can vary its tiling strategy across kernels while ensuring that the tile primitive to be loaded and reused on-chip is no more than 32 rows long and 32 columns wide. This square tile simplifies the SRAM organization, which will be explained in Section [3.4].

The current Scratchpad capacity has been set to 2 MB (mega-bytes) with a clock frequency target of 700MHz. The largest data type supported by the Scratchpad is Float16/BFloat16, with potential support for Float32 and Int8 in upcoming iterations of the Atalla family. The Maximum Vector Length (MVL), identical to the Maximum Tile Size supported by Atalla Ax01, is 32.

Internally, the Scratchpad is broken into 3 major components: "Frontend", "Backend", and "Body". Derived as a pipeline starting at the "Top" through the "Middle" and into the "Bottom", the Body abstracts away the SRAM controlling and interconnect coordination. The Frontend is responsible for communicating with the Vector Core, and maintaining a pipelined II=1 dataflow. The Backend is the DMA unit tasked with asynchronously working towards loading tiles from DRAM onto the chip. The Body is a black-box to both Frontend and Backend, ensuring that their logic is independent of the Body's internal timing.

Within the Body, the Scratchpad Top is a sequential structure that handles incoming upstream requests, propagates downstream stalls and deals with priority arbitration between the Backend and Frontend. To elaborate, upstream is the Frontend and Backend while downstream stalls could be potential SRAM glitches or Vector Core backpressure. The Scratchpad Bottom is a simple combinational response path that routes data to the correct requestor. Finally, The Scratchpad Middle is where the SRAM arrays and Crossbars (nicknamed "XBAR") live in an efficiently pipelined path that guarantees identical read/write latencies.

Modularizing components in this way helped to define clear boundaries and generate efficient test plans. Moreover, these layers of abstraction helped encapsulate responsibilities like valid/ready timing and protocol handling with other units of the accelerator core. The Scratchpad is fully parameterizable to realize a range of configurations. Its tunable parameters include total SRAM capacity (in bytes), number of banks, supported data-types, and crossbar topologies. Other knobs exist, but expose only register-transfer-level optimizations. Sub-units are supplemented with clever performance counters to assist in performance modelling and microbenchmarking of workloads to help evaluate the best core configuration for a specific workload. In the sections below, the architectural decisions that went into each Scratchpad sub-unit will be discussed.

## ***Section 3.2: Satisfying Systolic Array and Vector Core***

Google's TPUs have extremely large Systolic Arrays – called Matrix Multiply Units (MXUs) – but are still able to load in weights and inputs densely (i.e. not sparse and with minimal padding). We hypothesize that this is possible because their custom Scratchpad system – called Local Unified Buffer – and XLA compiler guarantee that intermediate operands are generated densely and in-order, maintaining the bandwidth required to keep the MXU fully utilized. Additionally, from our understanding of their Vector Cores, we believe that vectors are read and shuffled from large tiles before streaming them into the MXU. This is the only logical algorithm we surmised that would avoid extremely sparse dataflows during CONVs and transposes.

With this motivation, Atalla Ax01 features a 32-wide Vector Datapath that is in charge of controlling the 32x32-Systolic Array, both responsible for parallel computations at a tile-level and vector-level respectively. This datapath has its own set of vector operations that use vectors from an attached Vector Register File, which gets filled by reading Scratchpad data. This "core" requires a dedicated memory-operation path so that it can run the vector operations while parallelly loading the next set of data. For this reason, it includes a Vector Load/Store (VLS) unit for abstracting away the communication with the Scratchpad Frontend.

Each processing element in the Systolic Array computes a multiply-accumulate using an activation flowing horizontally and a weight flowing vertically. It is essential that both activation and weights are fed to the Systolic Array in strict lockstep. To guarantee full throughput, both operand streams must be delivered with cycle-accurate timing so that the next "wave" of data enters the Systolic Array simultaneously. Due to this, the Scratchpad must provide to the Vector Core two fully independent load paths, each sourcing from a different SRAM array, so that the activation and weight vectors can be fetched in the same cycle. Without synchronized arrival of the two, the Systolic Array cannot deliver continuous throughput, and the compute fabric would waste a significant fraction of available cycles.

## *Section 3.3: Interfacing with the AXI-Bus*

The maximum number of bits that can be transferred from the DRAM is 64. This constraint on channel width is established by the funding for this project, limiting the number of DIMMs to just one. Each element within the MVL is a 16-bit floating-point number. This limits our request/response packets to/from the AXI Bus to only 4 elements per slot.

With these limitations in mind, the Scratchpad Backend had to be created to split each vector into a maximum of 8 requests and a minimum of 1. Each request would handle the read/write requests/responses 64 bits at a time until the vector is exhausted. Read requests were largely unaffected by this limitation, as they are solely controlled by the Backend, and the only challenge was keeping track of where in the column the current request is. This led to the creation of UUIDs and sub-UUIDs. The UUIDs keep track of the vector we are transferring, and the sub-UUIDs keep track of the request we are processing. However, the read responses and write requests had to change.

Originally, the idea was to have separate read/write queues. There would be a queue for DRAM reads, DRAM writes, SRAM reads, and SRAM writes. The motivation was that, since writes and reads and requests and responses differed greatly, it would be easier to handle them in different queues. This idea was largely abandoned as the Scheduler would only issue either a Scratchpad Load or a Scratchpad Store at a time, and 2 queues would go unused, wasting space. Following this, the read and write queues were combined into a DRAM-Request-Queue and an SRAM-Request-Queue. Eventually, the decision was made to reduce the SRAM-Request-Queue to a single latch, as SRAM responses will happen quicker than the DRAM read responses can feed it, eliminating the need for a larger queue and saving area in the Backend.

Now with the DRAM-Request-Queue complete, we can latch our SRAM read responses, which at maximum will contain 512 bits. The data in the head of the queue will then be sent 64 bits at a time for our DRAM write requests. Counters had to be added to track how many requests had been completed and when to pop the head and move on to the next vector, or to stop if all requests for a vector had been sent. Additionally, the DRAM write request now needs a DRAM vector mask. Even though every request is 64 bits, that doesn't necessarily mean all 64 bits are useful.

For the DRAM read responses, each 64-bit packet is placed into the SRAM-Write-Latch. Again, an internal counter had to be created to ensure the timing of latches. The SRAM read requests aren't placed in the SRAM latch because the Scratchpad Body prioritizes Backend requests, and read requests can be created in a single cycle. A full state machine was considered over the use of counters. However, with the matrix tile size varying from 1x1 to 32x32, and the operation being simply tracking timing, counters were used instead.

The reason the SRAM needs only a latch while DRAM needs an entire queue is because of the expected memory latencies. DRAM read response latencies will be slower than the Backend prioritized SRAM write latencies, leading to the latch popping the write request before another DRAM read response comes in. The DRAM write requests, however, are expected to be slower than the SRAM read responses. It is possible that an entire SRAM read request will be processed and taken to the Backend before the previous response is able to send its 8 DRAM write requests. Thus, a queue was built for the DRAM request. Currently, its depth is 32 to store all 32 SRAM read responses at once.

Overall, these challenges were overcome through the use of counters and queues/latches. The counters help ensure timing. The DRAM queue allows the Backend to store SRAM read responses without worrying about replacing data. The SRAM latch helps the Backend build up the DRAM read responses until the row is ready to be sent out.

## *Section 3.4: Internal SRAM Organization*

The on-chip SRAM arrays need to be wide enough to accommodate 32 reads or writes in parallel, with each value being 16 bits. Simply put, the organization has to satisfy the bandwidth of 512 bits per cycle. Moreover, as explained in Section [3.2], there is a need for two static paths to exist, for a total of one-thousand-and-twenty-four bits per cycle bandwidth.

However, each path needs its own set of request-response and load-store units to guarantee the required asynchrony. Thus, we designed a parameterizable "Body" for the Scratchpad to allow for data-flow duplication and automatic interface alignment. Parameterized queues are placed where needed, in order to ensure II = 1 with the pipelined design. These queues handle backpressure in the uncommon case. Moreover, performance counters are embedded into the hardware for future modelling purposes. Each body handles its own set of statically assigned 512  bits.

Within a Scratchpad Stomach, there exists the SRAM arrays (called banks) which are controlled by a lightweight I/O controller. Our RTL design allows for many unique banking layouts depending on user-defined configurations. One could fix the bank's width or height, and the simulator or synthesis tool will infer the rest. One could instead define the desired aspect ratio (AR), commonly preferred to be between one and three-halves, and the design will automatically infer the other values. To make this possible, the bandwidth requirements are made constant. However, certain limitations exist with the current architecture, detailed in Section [7.1], that limits the explorable design space.

A naive layout would instantiate 32 independent 16-bit wide SRAM banks, satisfying the temporal constraints. However, for a total Scratchpad size of 2 MB, this would imply that the height of the array would be over ten-thousand entries, while the width stays at two bytes. This will result in long wordlines, high parasitic capacitances along the bitlines and large sense-amplifiers. From a VLSI perspective, this'll incur a severe delay from bitline precharging to bitline sensing, preventing the design from meeting timing requirements. To avoid this, we define the following equations that derive from the base parameters of (1) per-bank word-width (BANK_WIDTH), and (2) "folding" factor that defines the number of equal subarrays a tall array can be divided into (FOLDING_FACTOR).

$$\text{BANK\_CAP} = \frac{2 * 1024 * 1024(bytes)}{\text{NUM\_BANKS}}$$

$$\text{BANK\_HEIGHT} = \frac{\text{BANK\_CAP(bytes)}}{\text{BANK\_WIDTH}}$$

*Figure 3.4.1: Scratchpad Architecture*

To keep the physical aspect ratio of each bank acceptable, the FOLDING_FACTOR can be set to develop a network of sub-arrays which are serially accessed individually. While all the banks activate for a single read, all the sub-arrays in a single bank do not need to be active. However, this would limit the read-write port of the sub-arrays to the data-type size.

$$\text{SUB\_ARR\_HEIGHT} = \frac{\text{BANK\_HEIGHT}}{\text{FOLDING\_FACTOR}}$$

*Figure 3.4.2: Scratchpad Architecture*

Thus, each unique layout in the chosen hyperparameter space is identified by (BANK_WIDTH, FOLDING_FACTOR). In Section [6.2], we synthesize each configuration using the PCACTI tool and discuss an evaluation of their tradeoffs.

## *Section 3.5: Swizzling Support*

To guarantee row-major and column-major traversal, the base XOR swizzle operation needed to be customized for our use-case. Every vector, whether a logical row or a logical column of a tile, needs to be re-arranged before placing it into the banks. Upon reading these values out, they need to be swizzled again before they are sent into the Vector Core. The following equations were developed for this situation. They were first implemented and verified using a custom emulator, before verifying at the gate level using a RTL simulator.

First, when reading or writing a logical row into the Scratchpad, the swizzle unit needs to know about the number of rows & columns of the tile, the base address at which to be placed in the Scratchpad, and the logical row index within the tile. With this information, three vector masks are generated combinationally, using the following functions – a Shift Mask to map a vector value (lane) to a bank index, a Slot Mask to map the index within each bank for every vector value and a Valid Mask to indicate which indices need to be accessed. Below, "b_row" represents the swizzled bank index, "s_row" defines the index (or "slot") within a bank and "v_row" is a bit value that indicates if there is valid swizzled data for a particular bank. "v_row" finds its use when the number of columns of a tile are less than the Maximum Tile Size (32). “l” is the physical bank index, “r_abs” is the base address of the tile, “N” is NUM_BANKS. “C” defines the number of columns of the tile; note that the number of rows of the tile is not taken into consideration here.

$$b_{\mathrm{row}}(\ell) = (\ell \oplus (r_{\mathrm{abs}} \bmod N)) \bmod N$$

$$s_{\mathrm{row}}(\ell) = r_{\mathrm{abs}}$$

$$v_{\mathrm{row}}(\ell) = \begin{cases} 1, & \ell < C, \\ 0, & \ell \geq C. \end{cases}$$

$$\mathrm{valid\_mask}^{\mathrm{row}}[\ell] = v_{\mathrm{row}}(\ell)$$

$$\mathrm{shift\_mask}^{\mathrm{row}}_{\mathrm{lane2bank}}[\ell] = \begin{cases} b_{\mathrm{row}}(\ell), & v_{\mathrm{row}}(\ell) = 1, \\ \mathrm{None}, & v_{\mathrm{row}}(\ell) = 0. \end{cases}$$

$$\mathrm{slot\_mask}^{\mathrm{row}}[k] = \begin{cases} s_{\mathrm{row}}(\ell), & \exists\, \ell :\ v_{\mathrm{row}}(\ell) = 1 \wedge b_{\mathrm{row}}(\ell) = k, \\ \mathrm{None}, & \mathrm{otherwise}. \end{cases}$$

*Figure 3.5.1: Row-major Swizzle Logic*

Next, when reading or writing a logical column, the same information is required. However, the functions change to skew the data across the banks. A single change to the index location across banks becomes the core of the architectural design enabling our workloads. Below, “r_abs” is this changing base address (incremented for every value in the vector), while “b_col”, “s_col”, “v_col”, “l” and “N”, retain similar meanings from above. “R” defines the number of rows of the tile; note that the number of columns of the tile is not taken into consideration here.

$$r_{\text{abs}}(\ell) = r_0 + \ell$$

$$b_{\text{col}}(\ell) = \big(c \oplus (r_{\text{abs}}(\ell) \bmod N)\big) \bmod N$$

$$s_{\text{col}}(\ell) = r_{\text{abs}}(\ell)$$

$$v_{\text{col}}(\ell) = \begin{cases} 1, & \ell < R, \\ 0, & \ell \geq R. \end{cases}$$

$$\text{valid\_mask}^{\text{col}}[\ell] = v_{\text{col}}(\ell)$$

$$\text{shift\_mask}^{\text{col}}_{\text{lane2bank}}[\ell] = \begin{cases} b_{\text{col}}(\ell), & v_{\text{col}}(\ell) = 1, \\ \text{None}, & v_{\text{col}}(\ell) = 0. \end{cases}$$

$$\text{slot\_mask}^{\text{col}}[k] = \begin{cases} s_{\text{col}}(\ell), & \exists\, \ell :\ v_{\text{col}}(\ell) = 1 \wedge b_{\text{col}}(\ell) = k, \\ \text{None}, & \text{otherwise.} \end{cases}$$

*Figure 3.5.2: Column-major Swizzle Logic*

***Section 3.6: Interconnect Design Space***

As analyzed in Section [2.3], crossbars are a fundamental component for implementing swizzling in the read and write datapaths. In our work, we've analyzed four design choices for the crossbar – Benes network with a separate Control Bit Generation (CBG) unit, Batcher-Banyan network, CLOS network, and Benes network with a ROM. We built these designs incrementally, sequentially identifying bottlenecks in each design that another solves.

First, the Benes network was chosen for its minimal O(N log N) switch cost of simple 2x2 switches. This network is fundamentally rearrangeable nonblocking, meaning it can connect any arbitrary permutation of inputs to outputs without collisions within switches. In specific, Benes network carries out the permutation in 2log2(N) - 1 stages with N/2 of 2x2 switches for each stage. Each 2x2 switch takes 2 inputs and sends 2 outputs, and the output is either in the same order of the given input (straight through) or in the opposite order (cross), depending on the control bit that drives the switch. Then, the outputs of each stage are connected to the inputs of the next stage, until the last stage generates the final permuted data. Here, the connection from the output to input between stages is predefined for each N value without specific calculation, which will be discussed in the micro-architecture chapter.

Benes network requires specific control signals to control each switch, which need to be known ahead of crossbar initiation. Performing this task in hardware is explored in the context of sequential processing units in older telecommunication archives, but has scarcely been revisited in recent times. Bernstein[11] defined a set of fast parallel algorithms to generate these bits, but was not targeting ASIC cycle-driven realization of them. However, we chose to attempt it, and developed the CBG unit. The algorithm includes inverse composition, recursion and an internal loop that makes it useful for lightweight microcontroller chips sitting before a complex network route. Implemented as a parameterized HDL logic, large mux trees are instantiated when synthesized to standard cells in layout. The benefit of Benes network is simplicity and the fixed connection between the stages, but the CBG overhead led us to consider alternate designs.

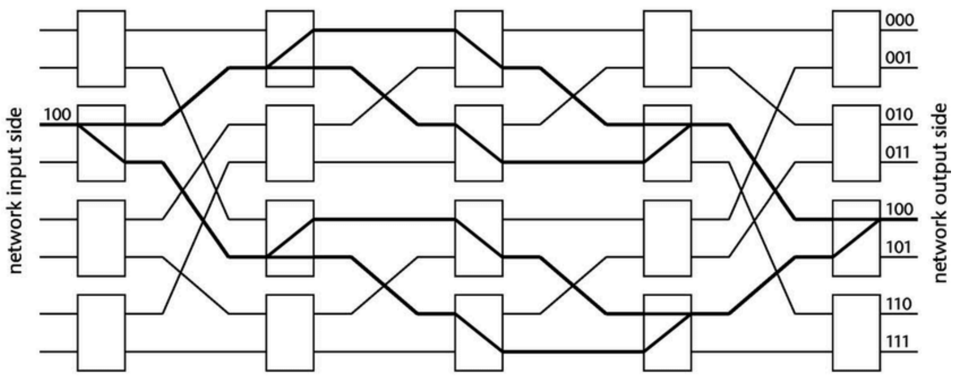


*Figure 3.6.1: Benes Switch Routing* [14]

The Batcher-Banyan network allows for making routing decisions at every permutation stage, eliminating the need of a pre-generated bitset. This network is based on the parallel Bitonic Sorting algorithm, which requires comparisons within every switch[12]. Thus, the complexity of each 2x2 switch increases with 16-bit comparators and the switch cost scales heavier as O(N $\log^2$ N). A Batcher–Banyan network with N inputs consists of log2(N) x (log2(N) + 1) / 2 stages, each containing N/2 of 2x2 comparing switches. We decided to implement this design to explore the tradeoffs with the comparator overhead. The main advantage of Batcher-Banyan design is the removal of the CBG block when all routing decisions are made locally in each 2×2 compare switch, but the drawback is a larger and more complex switch fabric. Batcher-Banyan requires more stages and more 2×2 switches, and each element is more expensive. This increases area, routing complexity, and critical-path delay through the network.

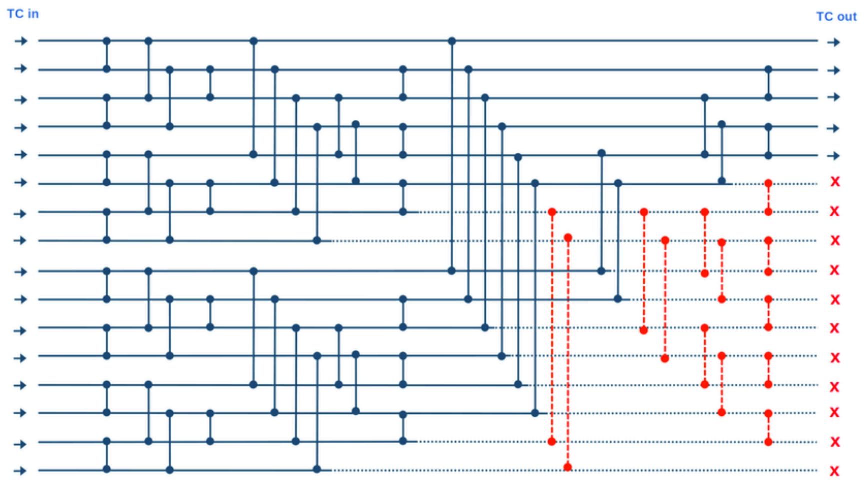


*Figure 3.6.2: Parallel Bitonic Sorting* [15]

After this, we decided to explore a design that inherently allows for dynamic routing decisions without the use of comparators – CLOS network. Heavily utilized in data centers for their large bisection-bandwidth, this design utilizes parameterizable switches that allow for less number of stages but makes the shifting logic more complicated. Note that while the compare logic is saved, the switch size itself increases to accommodate non-conflicting traversal paths. The logic enabling CLOS network is explored in great detail in Section [4.4]. The benefit here is the absence of CBG modules and a shorter per-packet latency, but hardware cost of each switch is slightly higher.

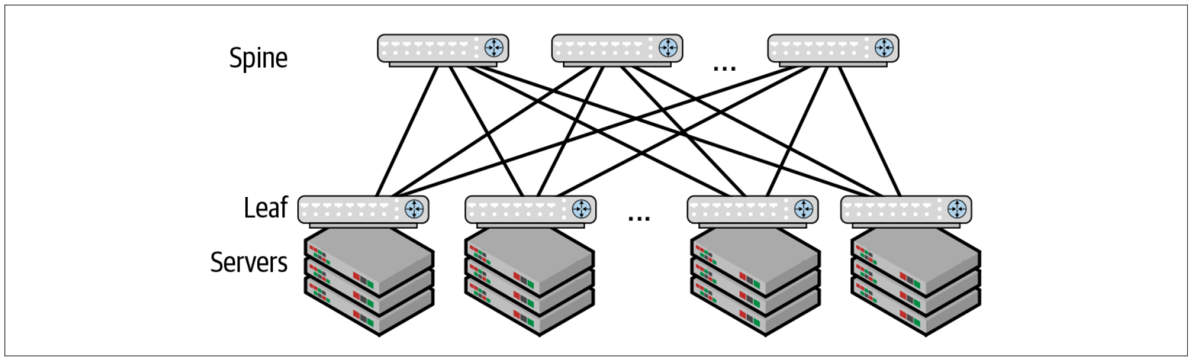


*Figure 3.6.3: (Folded) CLOS Networks in TOR Switches* [16]

This design is a generalized version of Benes network and consists of 3 fixed stages regardless of the number of inputs. There is an input stage, a center stage, and an output stage, and each stage includes a specific number of switches. The number and size of switches are identical for input and output stages, and the arrangement of the center stage is unique. The CLOS network can be designed as rearrangeable non-blocking (RNB) or strict non-blocking (SNB), depending on how the three stages are parameterized[13]. Here, non-blocking means that all the input and output ports can be connected one-to-one for each stage without overlap. SNB is where the ports can always be non-blocking with pre-existing wire connections, whereas RNB may require some rearrangement of the wires to achieve non-blocking assignments. The parameters of the CLOS network are defined as follows: C(n,k,m), where n is the size of input and output modules, k is the number of input and output modules, and m is the number of center modules. The requirements for CLOS network to be considered a SNB is $m \geq 2n - 1$, and for RNB is $m \geq n$. Here, RNB is more efficient, because of less hardware cost of the center module.

After synthesizing the above designs, we preferred the Benes network for its low area, but were constrained by the CBG requirement. We hypothesized that the regularity of the permutation masks should also map to the dynamically generated control bits. As such, we should be able to identify all the unique bitsets and fix them into a ROM instantiated next to the Benes. Before the Benes network is triggered to consume an incoming request, the ROM can be accessed to get the required set of control bits. However, this would require that every possible permutation, with its producing metadata, has to be uniquely mapped to every ROM entry. In the following section, we explain the experiments that lead to achieving our goal and finalizing the Benes + ROM design as a viable option. However, further studies are needed to concretely evaluate whether ROM size is better than the CBG overhead.

## ***Section 3.7: Replacing the CBG with ROM***

In order to identify all the dynamically generated permutations of these masks, the emulator was refined to track unique values and their producing configurations. Through those experiments, another set of equations were identified to uniquely map swizzle inputs to a set of masks. This motivation for this work was outlined in Section [3.6], under the Benes + ROM design choices, and helps statically map the behaviour of the swizzle unit into a fixed set of masks.

When reading/writing a logical row from/to the Scratchpad, we use the logic from Figure [4.5.3] to identify the ROM Index. When doing the same for a logical column, we use the equations in Figure [4.5.4], wherein the midpoint of the bank index space ("H") is used to enable column remapping while maintaining the uniqueness of the produced ROM index. “LOG2B” defines the number of bits needed to represent all the bank indices, while “H” is half the NUM_BANKS value. Both these ideas can be realized with the addition of a few gates to the swizzle unit, adding minor delay to the signal propagation.

$$\begin{aligned}\text{abs\_row} &= \text{base\_row} + \text{row\_id} \\ \text{rom\_index} &= \text{abs\_row} \,\&\, (\text{NUM\_BANKS} - 1)\end{aligned}$$

*Figure 3.7.1: Row-major Swizzle - ROM Indexing*

$$\begin{aligned}\text{base\_low} &= \text{base\_row} \,\&\, (\text{NUM\_BANKS} - 1) \\ \text{base0} &= \text{base\_low} \,\&\, (H - 1) \\ \text{msb} &= \big(\text{base\_low} \gg (\text{LOG2B} - 1)\big) \,\&\, 1\end{aligned}$$

$$(\text{canon\_base},\ \text{canon\_col}) = \begin{cases}(\text{base\_low},\ \text{col\_id}), & \text{if msb} = 0 \\ (\text{base0},\ \text{col\_id} \oplus H), & \text{if msb} = 1\end{cases}$$

$$\text{rom\_index} = (\text{canon\_base} \ll \text{LOG2B}) \mid \text{canon\_col}$$

*Figure 3.7.2: Column-major Swizzle - ROM Indexing*

From our experiments, with a maximum tile size of 32, we were able to identify 512 unique CBG Bitsets (each 144 bits) to be stored at boot time into the ROM. We also noticed that many of the lower bits of these entries (64 of the 144 bits) were dominantly zero’d, allowing us to optimize the ROM size by masking off some of the Benes crossbar’s control bits. As such, we’d need a 5 KB ROM to store these values and guarantee a single-cycle latency.

# Chapter 4: Micro-Architecture

## *Section 4.1: Scratchpad Body*

The largest components of the Scratchpad are grouped into one large unit called the Body. As a whole, it is responsible for accepting requests from the Frontend and Backend, routing data into the correct read/write path, controlling the SRAM banks and routing responses back. These "requests" and "responses" are packetized into bundles of metadata and vector bits in the RTL design, simplifying their implementations.

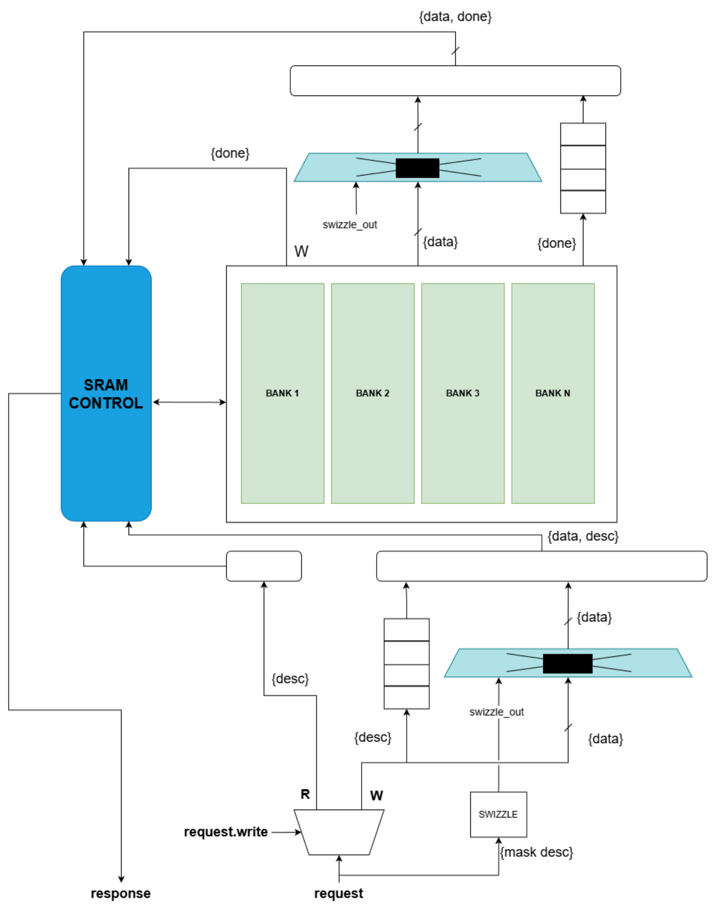


*Figure 4.1: Scratchpad Body Diagram*

The "Middle" portion of the Body deals with the crossbars and SRAM. The incoming requests get demuxed based on a "write" bit in the packet. For read requests, the request gets sent directly downstream to the Body's SRAM Controller. The swizzled data is fetched from the SRAM banks and sent to the Read Crossbar, for re-swizzling back to the original order. For write requests, the write data gets routed directly to the Write Crossbar, because the data needs to be swizzled before the SRAM controller begins accessing the macros. Alongside each Crossbar, a FIFO is instantiated to propagate request metadata while it rearranges the vector values.

The "Top" unit arbitrates requests between the Frontend and Backend. The Backend, outlined in Section [4.3], is carefully optimized to combinational logic. Moreover, it interacts with the Body sparsely over time, owing to the DDR4 protocol latencies and general DIMM timing overheads. Thus, as the uncommon case, priority is given to the Backend to guarantee that the path to the AXI Bus is never stalled. When the "grant" for a cycle is allocated to the Backend, the Frontend path is completely stalled. The Vector Core sees a signal go high, informing it to latch and hold its current outgoing packet. Additionally, FIFOs are added near the Top to ensure backpressure situations with the (extremely uncommon) glitching in the SRAM macros.

Finally, the "Bottom" unit routes the response to the requesting unit. This information is contained within the packet through an identifier. The response paths in the Frontend, and consequently Vector Core, are simulated to never stall. The Vector Core contains additional latches to handle internal backpressure. The response path in the Backend, and consequently AXI Bus, are guaranteed to never stall. This is confirmed by a relatively larger FIFO, and the policy within the Bus's protocol that prioritizes the Scratchpad over other managing units.

## *Section 4.2: Achieving Vector Core Requirements*

Each Frontend unit is one-read-or-write ported — that is, only one request can be serviced per cycle. It consists of combinational logic to communicate with the Vector Core and swizzle units for packetizing metadata to send towards the Body. Independent of whether it is a read or a write request to the SRAM banks, the generated swizzle mask by the Swizzle Unit is the same, owing to the inverse nature of the function.

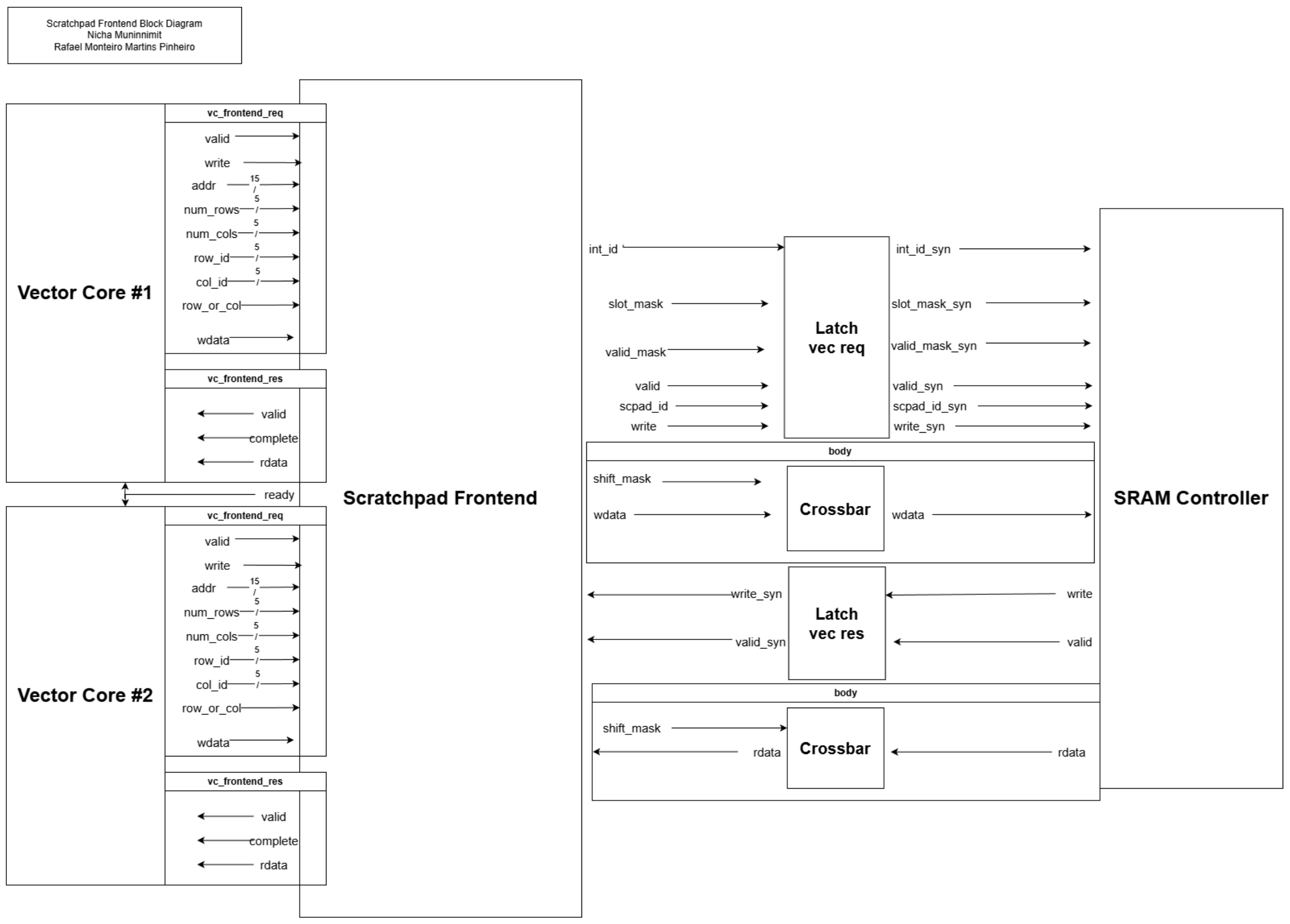


*Figure 4.2: Frontend Top-Level Diagram*

## *Section 4.3: Designing a Non-Blocking Backend*

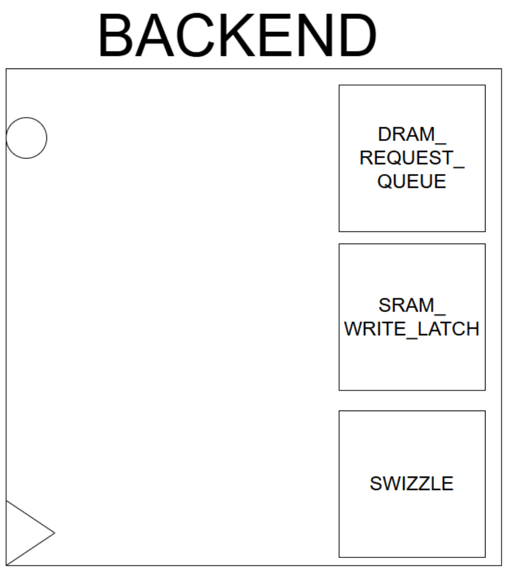


*Figure 4.3.1: Backend Unit Top-Level Diagram*

The Scratchpad Backend module features a DRAM-Request-Queue, SRAM-Write-Latch, and a Swizzle unit. The Scheduler Core will send the Backend a request packet (a logical bundle of bits) that includes the base addresses (Scratchpad and DRAM), matrix metadata (height and width), and the read/write signal. Once a request is received, each row request is split by left-shifting the number of columns by 2, and each 64-bit request is given a DRAM vector mask. The common case for the mask is to use all 64 bits. However, when the last request is detected, the bottom 2 bits of our matrix column size are checked, and the DRAM vector mask is created. For example, if a DRAM request is made for a 6-element vector, then the first request will need to use the entire 64-bit bus and produce a mask of 1111, but the second/final request will only use 32 bits and a mask of 0011.

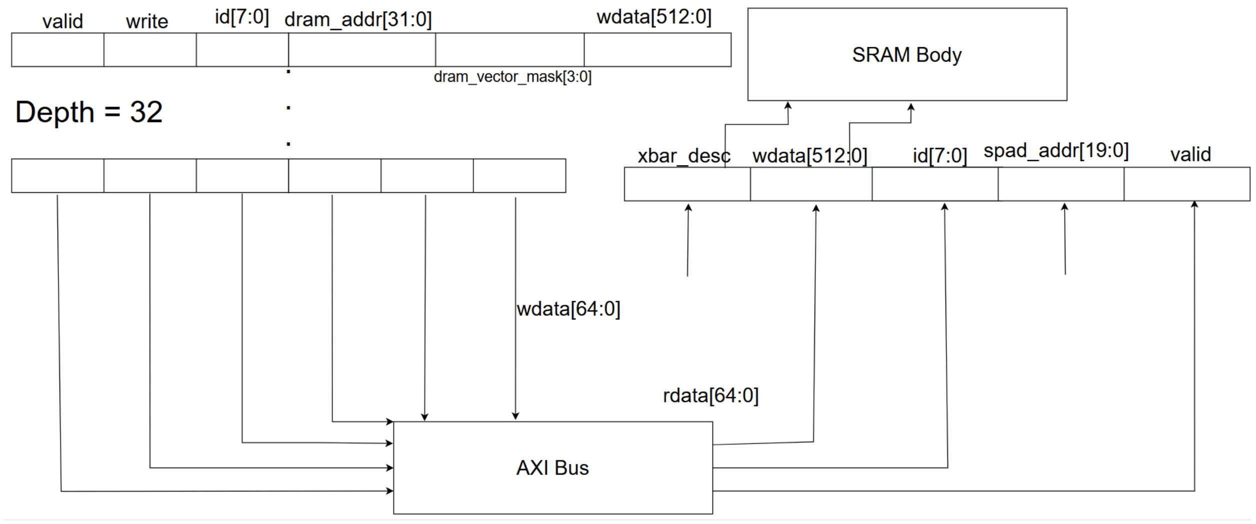


*Figure 4.3.2: Scratchpad Load Flows*

If the request was a Scratchpad Load instruction, then the Backend begins generating DRAM read requests and placing them into the DRAM-Request-Queue. The DRAM address is calculated based on the base DRAM address from the scheduler and the row/request we are currently on. The row and request are determined by the UUID produced in the Backend, with the lower 3 bits representing the request we are on and the upper 5 bits representing the row. The UUID and sub-UUID are counters that continue until they hit their respective number of rows/requests. In the DRAM-Request-Queue, whenever a request is completed, an internal request completed counter is incremented, and a signal is sent out to the Backend to increment the sub-UUID. When the request completed counter hits the number of requests needed, a transaction complete signal is sent to the Backend, and the UUID is incremented. If the UUID matches the number of rows and the sub-UUID matches the number of requests, then we know the final row and request have been created, and the initial read request can stop being created.

At the same time, the SRAM-Write-Latch is waiting for DRAM read requests to be received. The SRAM-Write-Latch will build our row request 64 bits at a time. The scratchpad address and its offset, what row we are currently working on, are given to the swizzle unit, and the crossbar description is created for our row transaction. The SRAM-Write-Latch has an internal request completed counter that will increment whenever a DRAM read request is completed and received. Once the request completed counter hits the number of requests needed the data in the latch will be sent to the Scratchpad Body, and a signal is sent to the Backend to increment its request completed counter. The Backend request completed counter will increment until it matches the number of rows, at which point it will send a signal to the scheduler indicating the Scratchpad load is complete. An extremely uncommon case for the Scratchpad load is when stalls are sent from the body or DRAM. For the DRAM stall, the head will no longer be popped, and requests will continue to be created until the queue is full. Since the read requests are entirely determined by the Backend, they will simply be paused, and the corresponding counters will no longer increment. If the Body is stalling the Backend, then the SRAM-Write-Latch will continue to build the write request until it is completed, at which point the latch will tell the Backend that it is full and to pause responses.

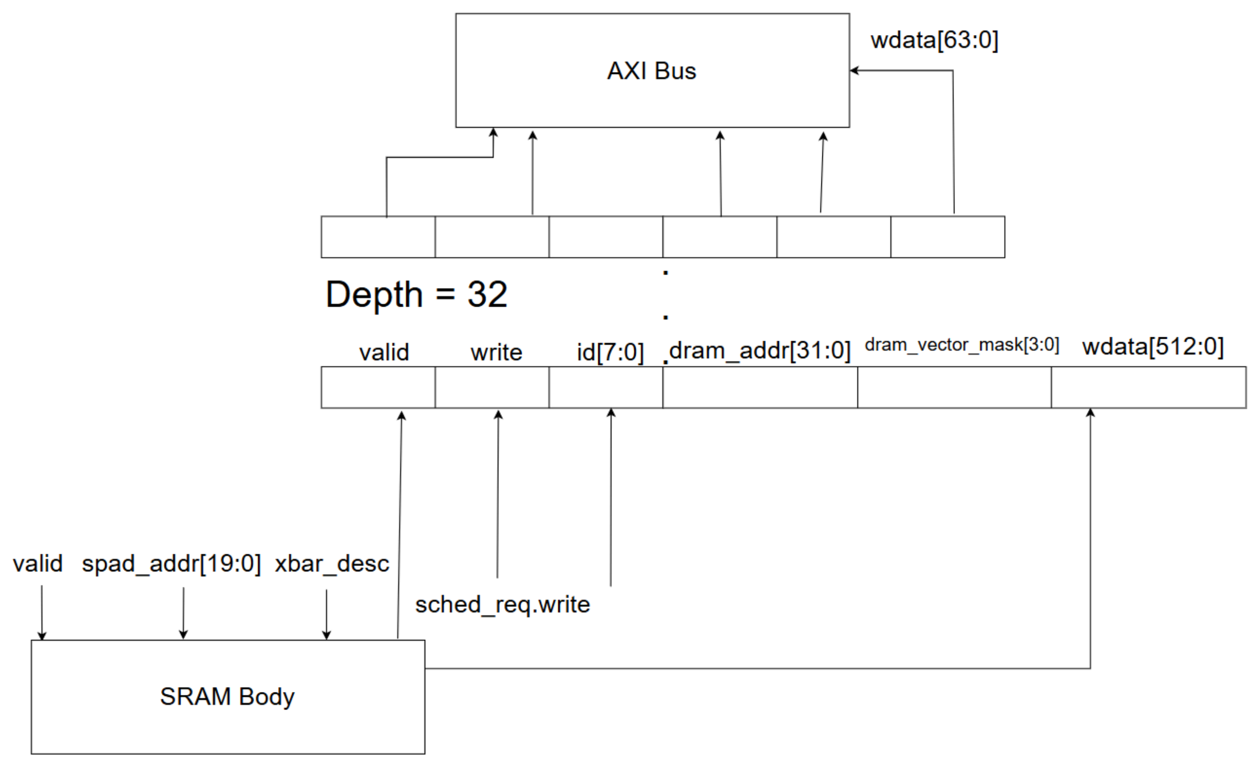


*Figure 4.3.3: Scratchpad Store Flows*

If the Scheduler request was a Scratchpad Store, then the Backend will generate the swizzle metadata and send it to the Scratchpad body. The Scratchpad address is calculated with the base address plus the UUID. When the UUID reaches the number of rows, the initial read request will stop being produced. As requests complete, they are latched in the DRAM-Request-Queue. Again, using the matrix size given by the Scheduler, the row is split into requests. The outgoing DRAM address is now calculated with the base DRAM address, plus the Backend request complete counter and the sub-UUID. The DRAM write request is kept in the head of the queue until all write requests for that row have been sent out, at which point the head is popped and incremented. Every time a request is sent, a burst complete signal is sent to the Backend to increase the sub UUID, and each time the head is popped, a transaction complete signal is sent to increment the Backend's request completed counter. Once the backend's request completed counter reaches the number of rows, and the sub-UUID reaches the number of requests, the Backend latches a signal and sends it to the scheduler to indicate the Scratchpad Store is complete. For the stalls, if a body stall is detected, the Backend simply stops generating SRAM read requests. For a DRAM stall, the head is simply not popped, write requests aren't generated, and SRAM responses can continue to be accepted until the queue is full, at which point it will tell the SRAM controller to stop sending responses.

## *Section 4.4: Crossbar Implementations*

In our common case of $N = 32,$ there are 9 stages and 144 of 2 x 2 switches, with each stage containing 16 switches for the Benes network. The structure follows the recursive pattern[11]: First, the inputs enter the switches paired with its immediate neighbor, creating $N/2$ groups of one switch each. In the following stages, the number of switches for each group doubles, and the total number of groups decrease accordingly, continuing until one group includes all the switches in the stage. After this stage, this process is reversed. The number of switches halves for each group, and the number of groups increases. An example of this is represented in Figure [4.4.1], using size of N=8 for effective visual visibility.

Moreover, the Control Bit generation (CBG) module generates the control bits which control each of the 2x2 switches. This means that for N=32, there will be 144 control bits to match the number of switches. The role of CBG is accepting shift masks as an input and generating the control bits. It operates on a logic from a well-defined software algorithm, which was translated into a hardware module.

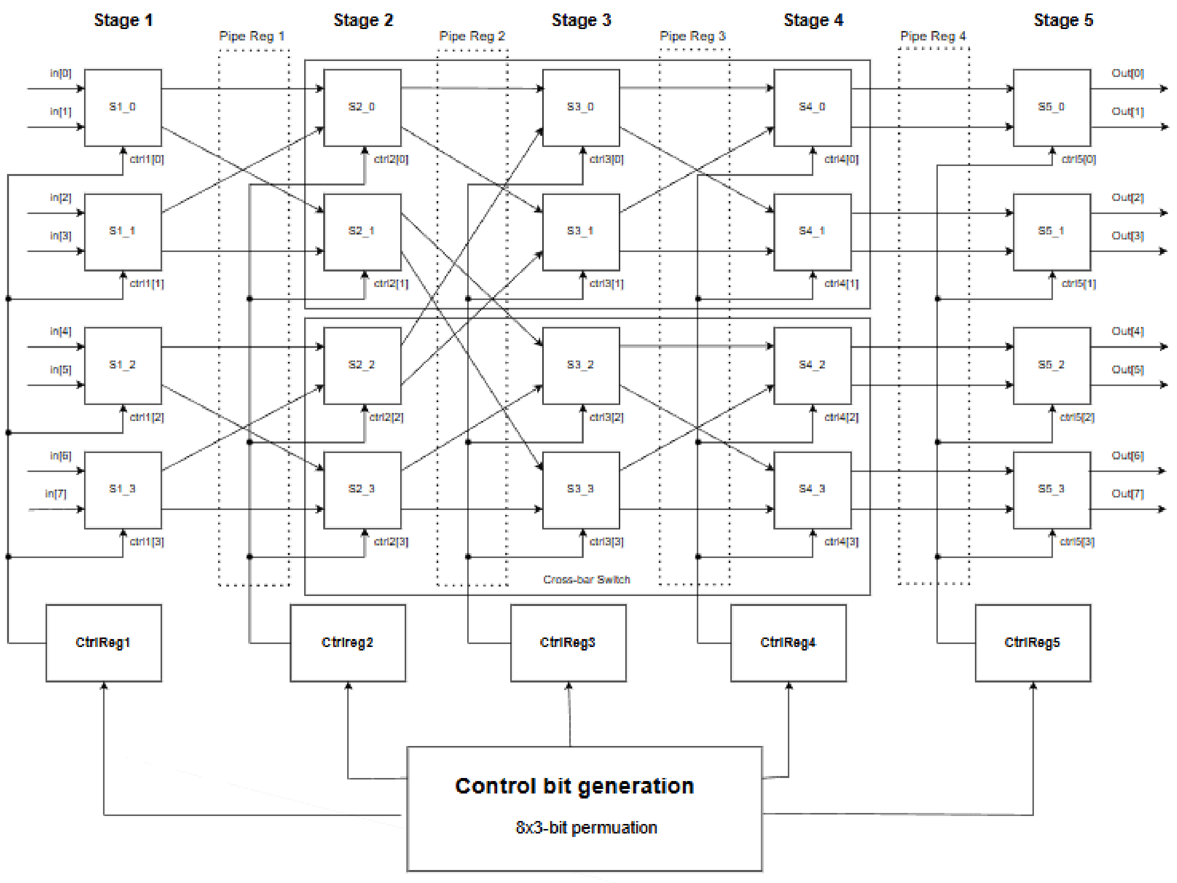


*Figure 4.4.1: 8x8 Benes network RTL diagram*

In contrast to the Benes network, which relies on precomputed control bits from a CBG module, the Batcher-Banyan network uses local compare & swap logic inside each 2×2 switch. Each compare switch takes a pair of data and shift mask, compares the shift masks of the two inputs, and decides whether to pass them straight through or cross them. In this way, the shift mask effectively acts as the key that drives the permutation, and the data and its shift mask always move together as a pair. For our common case, there are 15 stages and 240 switches in total. The stages follow the Batcher pattern: similar to Benes network, inputs are compared within smaller local groups, and as the network advances through the stages, the group size doubles, until the group size reaches the maximum. Then, the group size halves until it reaches the minimum of one switch each. In

Batcher-Banyan, the group size repeats two times as it increases but only occurs once after it reaches the maximum and decreases. The output-input wire connection follows a very specific pattern within each group[12], as shown in Figure [4.4.2].

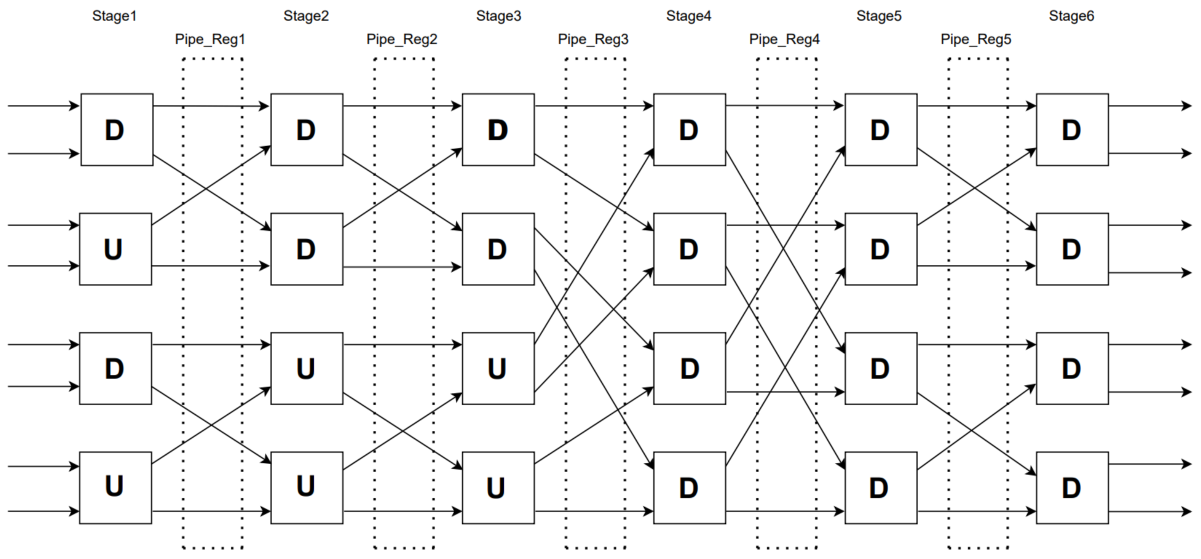


*Figure 4.4.2: 8x8 Batcher-Banyan network RTL diagram*

The third design is the CLOS network. Considering RNB and the hardware cost, the most efficient design of the CLOS network for N=32 is C(4,8,4). This means that there are 8 input and 8 output modules of size 4 each, with 4 center modules of size 8, as shown in Figure [4.4.3]. The input data will be entered into 8 different input modules sequentially, then they enter the center modules based on the target output module. Each center module holds the data such that its index corresponds to the target output module. If a center module of the desired index is already filled by previous data, then the next module is selected instead. After the center module connection is done, these data are directly connected to each target output module. Lastly, each output module sorts and outputs the sorted data with a simple crossbar. Because connection rearrangement is done based on the associated shift mask, the input is given as a data and shift mask pair, just like the Batcher-Banyan network.

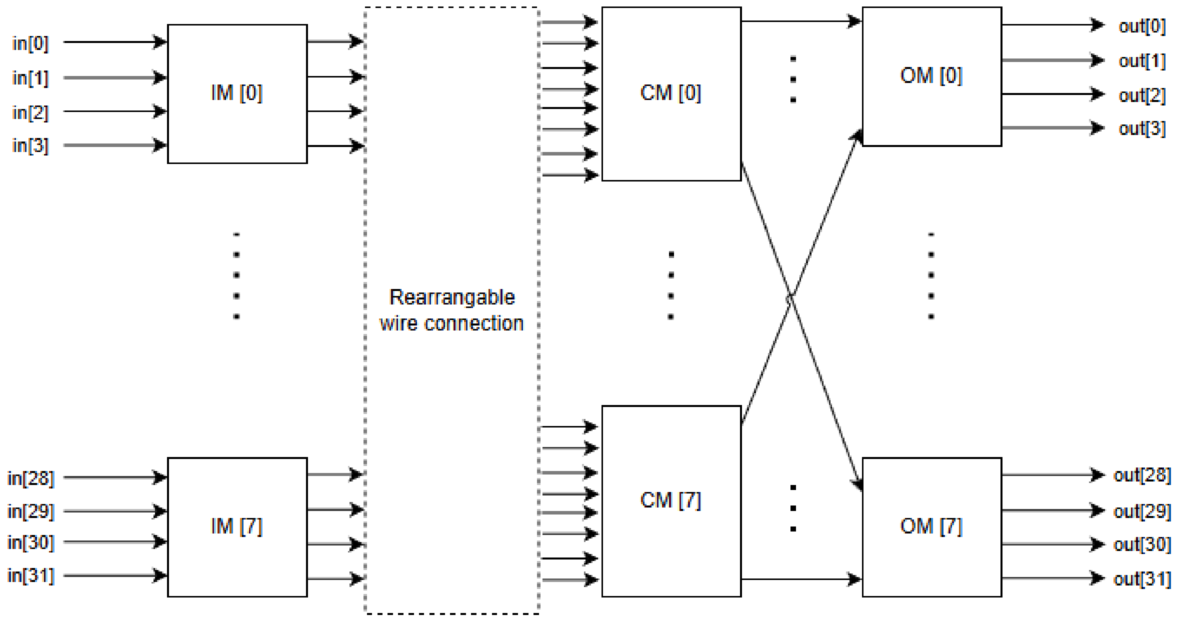


*Figure 4.4.3: 32x32 CLOS network RTL of C(4,8,4)*

The final implementation choice is using Benes with ROM. As explained in Section [3.7], there will be 512 unique control bit combinations. This means that it is possible that the Benes network can be driven without the CBG module, reducing the cost of CBG while keeping the throughput of the Benes switch network. Here, using the entire shift mask to search the memory will require usage of a large number of bits, so specific row and column indexes are used instead, minimizing hardware cost of ROM.

All of the designs are parameterized and pipelined to allow testing with different parameters. For example, Benes network and Batcher-Banyan network were tested with different numbers of cycles, from fully-pipelined to single-cycle configurations to determine the most efficient design. The crossbar also needs to do permutation of non-common case of $N \neq 32$, which adds the need for parameterization.

# Chapter 5: Programming Model

## *Section 5.1: Software Approach*

The Ax01 Programming Model allows users to map C-based algorithms to our throughput-focused accelerator, architected around VLIW vector-datapath, software-managed Scratchpad and a 32x32 BFloat16 Systolic Array. Unlike GPUs, Atalla Ax01 does not expose a wide SIMT/SIMD programming interface. Instead, it provides a tile-centric compute model where kernels explicitly optimize and orchestrate data movement, vector lane utilization and systolic array computations through a unified instruction stream. The representative workloads need both wide-and-deep pipelines specifically for two-dimensional matrices.

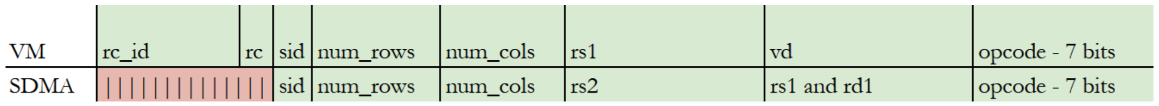
As explained before, Ax01 is not a general-purpose processor. It is a core that will be placed alongside a high-performance CPU for a heterogeneous compute platform. Programmability is offered through C-based intrinsics and a custom compiler toolchain. We do not plan to support imperative languages like Python. The core will be programmed using a host-device model, similar in spirit to AMD's HIP but fundamentally simpler [8].

Host responsibilities include allocating tensors in DRAM, launching device kernels, passing tile descriptors and kernel metadata. Device responsibilities include moving tiles between DRAM and on-chip SRAM, swizzling data within the Scratchpad, loading vectors into the Vector Datapath's Register File and executing blocking vector load/store/compute operations. The compiler issues VLIW bundles into a partition of the DRAM address space, privately exposed to the device. The host will write certain flags and kernel metadata to a pre-defined location in the DRAM, which wakes the device to complete its tasks.

The Atallax01 memory system is software-managed, and does not enforce any hardware-managed ordering mechanisms. The datapath is in-order, with the DMA instructions making Scratchpad locations valid before later accesses take place. Global Memory (GMEM) is the formal name assigned to the DRAM chips, which are high latency accesses and hold the large multi-tile multi-channel tensors defined by the host. The Scratchpad Memory (SCPAD) is the low-latency 2MB on-chip SRAM, accessible via Vector Memory and SDMA instructions defined in Section [5.2]. The Vector Register File (VREG) is also accessible through the Vector Memory instructions, and is a SRAM array logically laid out as wide registers instead of cache lines. The Systolic Array also utilizes some accumulation buffers to store the transient outputs, but is not programmable; it is a purely hardware-coordinated unit.

***Section 5.2: Compiler Support***

The Atalla's PPCI-Compiler is set to support two sets of memory load-store instructions – Vector Memory (VM) and Scratchpad DMA (SDMA). All the unique instructions fit within a 40-bit specification sheet and are identifiable by a 7-bit opcode, as shown in Figure [5.2.1]. "rs1", "rs2" and "rd1" are 8-bit operand indices for scalar registers embedded into the instruction encoding, while "vd" is the 8-bit index for a vector register. "rc_id" identifies the row or column index while the "rc" bit chooses which traversal ordering to follow. "num_rows" and "num_cols" inform the Scratchpad about the tile size with the "sid" mentioning which set of SRAM arrays need to be accessed.



| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| VM | rc_id | rc | sid | num_rows | num_cols | rs1 | vd | opcode - 7 bits |
| SDMA | 🟥 | 🟥 | sid | num_rows | num_cols | rs2 | rs1 and rd1 | opcode - 7 bits |

*Figure 5.2: Ax01 Bit-Specification*

The on-chip scheduling unit enforces a tainted-VLIW scheme by checking inter-bundle dependencies through scoreboarding, while the compiler ensures intra-bundle independence. Moreover, the 4-slots within a bundle are constrained in the number of instruction permutations they can fit. For example, a bundle can only contain 2 or less SDMA instructions, owing to the duplicated datapaths explained in Section [4.2]. According to elaborate emulation beyond the scope of this report, the number of unique bundle permutations is capped at 18,000.

# Chapter 6: Results

The work accomplished by this team involves implementing and verifying sub-units of the Scratchpad. The end goal of the Atalla initiative is to tape-out a custom accelerator on MIT-LL's 90nm process, potentially switching to the TSMC 65nm node in the future. As a first iteration, the verified sub-units were synthesized on this open-source technology using licensed Cadence Genus and Innovus toolchains, while the SRAM banking strategies were approximated using the PCACTI tool. PCACTI is an updated version of Hewlett Packard's cache modelling tool, CACTI, which supports necessary features like vector registers.

## *Section 6.1: Backend Verification*

| Module | Branch | Statement |
|---|---|---|
| backend.sv | 100%<br>(36/36) | 100%<br>(100/100) |
| dram_request_queue.sv | 100%<br>(23/23) | 100%<br>(72/72) |
| sram_write_latch.sv | 100%<br>(8/8) | 100%<br>(20/20) |
| swizzle.sv | 33.33%<br>(2/6) | 57.69%<br>(15/26) |

*Table 6.1: Backend Code Coverage*

The Backend was verified using simulated Scheduler, SRAM Controller, and AXI Bus Interfaces. The tests were created in a SystemVerilog testbench. Assertions were formulated to detect if the signal timings were correct and if the expected data and addresses were found. The simulated units were presumed to respond every cycle due to a lack of clarity on their internal decision-making mechanisms

To achieve 100% code coverage across all the Backend design files, every tile size combination was tested for Scratchpad Load and Store sequences. The corner cases in the Scratchpad address ranges were also tested. Every stall was tested, and the conditions for a full queue and full latch were tested. All the tests passed the automatic checker. The only concern is the Swizzle unit. The Backend is using the general Swizzle unit created for the Scratchpad; however, the row or column feature of the Swizzle unit is unnecessary. The Backend only creates row major requests. In the future, it should be considered to design a special Backend Swizzle unit to reduce some area and increase code coverage.

## *Section 6.2: Synthesized SRAM Networks*

The following visualizations show us the trend of varying FOLDING_FACTOR across a fixed Scratchpad capacity (2MB), BANK_WIDTH (16bits) and NUM_BANKS (32).

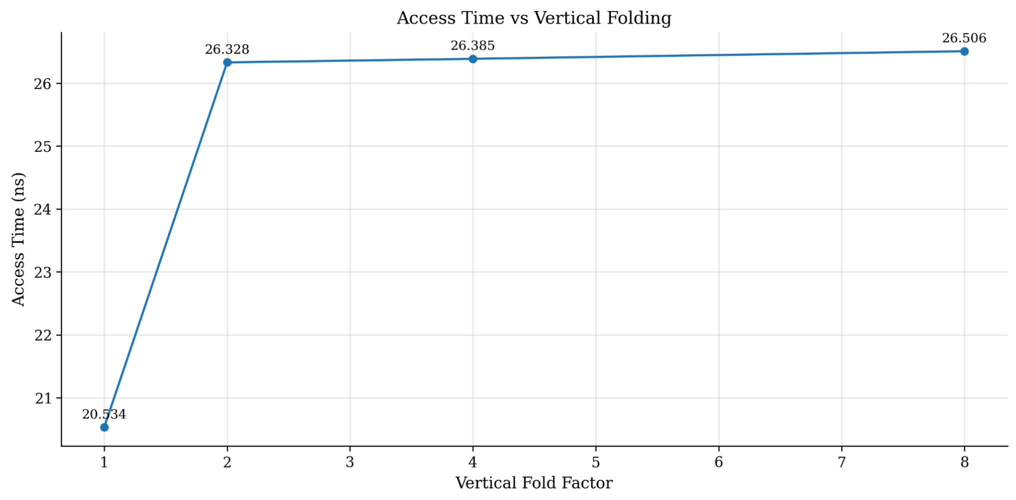


*Figure 6.2.1: Access Time (ns) trend with increasing folding factor*

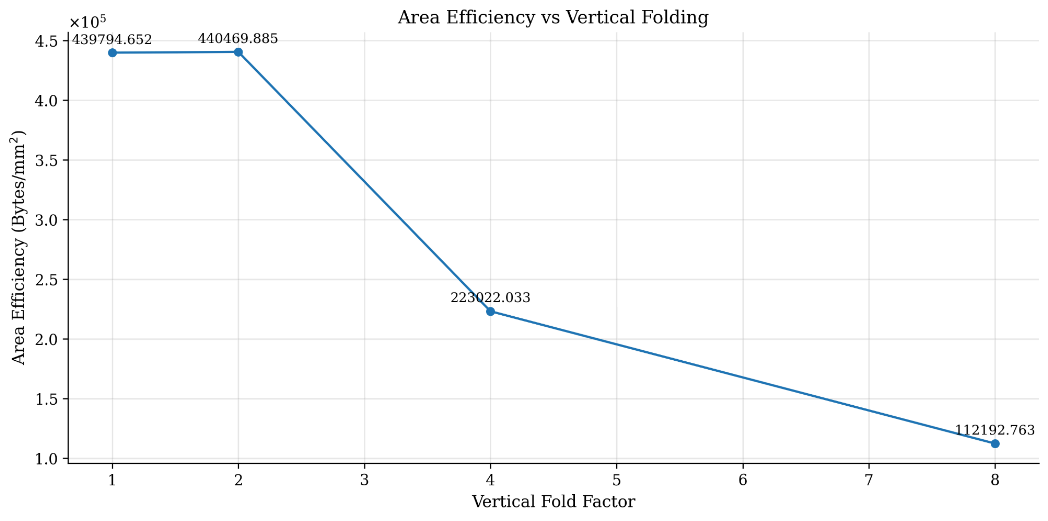


*Figure 6.2.2: Area Efficiency (Bytes/ $mm^2$) trend with increasing folding factor*

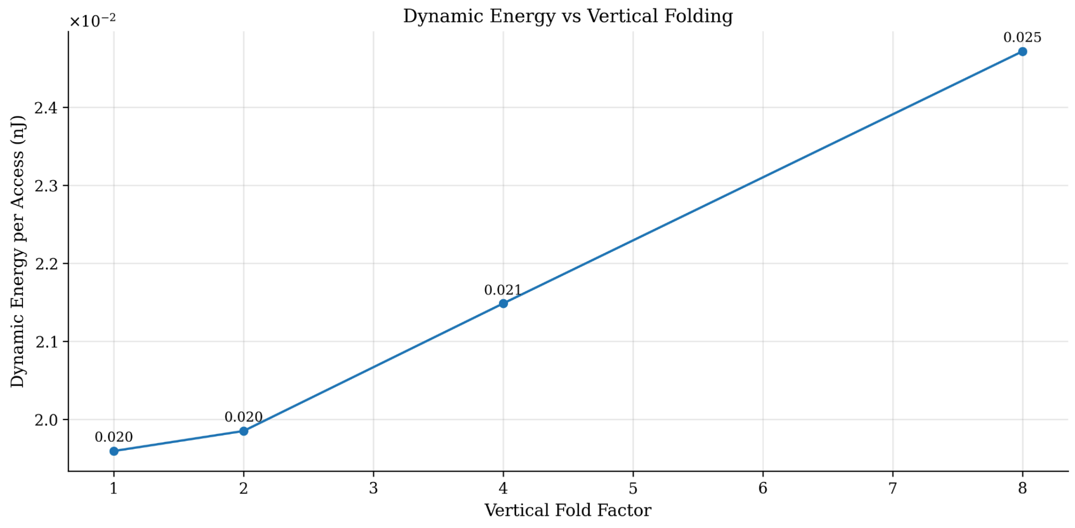


*Figure 6.2.3: Dynamic Energy (nJ) trend with increasing folding factor*

From the above figures, we notice a clear trend of worsening macro characteristics – specifically energy, area efficiency and delay. We attribute this to the wiring overhead required to achieve sub-banking while maintaining the same read-write port size. Each new sub-array requires its own local set of decoders, sense-ampliers and drivers on top of the existing global indexing logic. From our analysis, sub-banking provides the greatest benefit when applied to wide arrays which get bifurcated and folded across configurations.

## *Section 6.3: Synthesized Interconnect Designs*

Below is the synthesis result of clock speed, total area, and power of different crossbar designs processed with MITLL 90nm.

| Design Name | Version | Clock Speed (MHz) | Total Area (mm2) | Power (mW) |
|---|---|---|---|---|
| Benes | 1 Cycle | 858.43 | 0.05 | 16.10 |
| | 3 Cycle | 1511.94 | 0.06 | 30.19 |
| | 5 Cycle | 2145.55 | 0.08 | 41.57 |
| | 9 Cycle | 2862.01 | 0.11 | 61.25 |
| | | | | |
| Batcher-Banyan | 1 Cycle | 146.97 | 0.39 | 401.32 |
| | 3 Cycle | 378.85 | 0.37 | 361.99 |
| | 5 Cycle | 580.59 | 0.36 | 354.523 |
| | 15 Cycle | 709.12 | 0.28 | 167.29 |
| | | | | |
| CLOS | 3 Cycle | 4088.31 | 0.21 | 56.61 |
| | | | | |
| CBG | 5 Cycle | 582.72 | 1.00 | 220.54 |

*Table 6.3: Synthesis result of Crossbar options with MITLL90nm*

According to Table [6.3], Benes network has the lowest total area for all the versions and power for all the versions except for 9 cycles, which is slightly lighter than CLOS network. Within the different versions of Benes, 1 cycle had the best result in all three areas of clock speed, total area, and power. Batcher-Banyan, on the other hand had much lower clock speed with higher total area and power for all the versions, from 1 cycle to 15 cycles. With the Batcher-Banyan network, the numbers are the best for 15 cycles. The clock speed was the fastest for the CLOS network with 4088.31MHz, and the area and power is between Benes network and Batcher-Banyan network. CBG clearly had the worst results, with a very high area of almost 1mm$^2$ and power.

After analysis, the Benes network is the most efficient compared to any other designs, considering all three of the parameters. The 1 cycle version was chosen to be the most efficient for Benes network, with outstanding numbers of more than 2 times smaller total area and almost 4 times smaller power consumption compared to the 9 cycle version (fully pipelined; II=1) version. Although the clock speed is much slower, this is acceptable for the purpose of the Scratchpad. Within the acceptable range, area and power takes more priority in determining the optimal solution. However, all three statistics of CBG are not ideal, with large total area and power. With addition of the Benes network, the total area becomes greater than 1mm$^2$ and power also is 236.64mW. This is more than 21 times the area and 14 times the power of the Benes network itself. The clock speed will be limited to that of CBG, and the latency will also be increased to be 5 cycles of CBG, limiting throughput. Because the cost of the CBG module just to drive a very efficient Benes network is too high, the design choice of using the Benes network with CBG will not be considered.

Batcher-Banyan network itself is not as efficient as Benes network by itself, but it does not require CBG, making it more applicable to the optimal solution. Although the three numbers are the best for 15 cycles version for the Batcher-Banyan network, its cycle latency is too high, which decreases throughput of the crossbar. Therefore, 3 cycles or 5 cycles were considered to be more efficient regardless of having lower clock speed and higher area and power.

The CLOS network improves the numbers of the Batcher-Banyan network. Comparing it with the Batcher-Banyan network version of the same latency of 3 cycles, its clock speed is 10.78 times faster, total area is about 1.78 times smaller, and power is 6.40 times smaller. Similarly, its clock speed is 7 times faster, area is 5.08 times smaller, and power is 4 times smaller, even if I compare it with the most efficient Benes network version of 1 cycle. Therefore, the CLOS network  is considered to be more efficient than Batcher-Banyan network and Benes network (with CBG), and is chosen to be the optimal solution for the crossbar design of the Scratchpad.

# Chapter 7: Limitations

## *Section 7.1: SRAM Banking*

The deep-dive in Section [3.4] outlines the potential SRAM banking knobs, but highlights a mandatory constant value for NUM_BANKS. Swizzling allows us to avoid bank conflicts, and read vectors in both row-major and column-major traversal. To support this, we need to guarantee that NUM_BANKS = MVL. The following visualization, in Figure [7.1.2], shows that swizzling values across the logical column of a tile will stagger them between SRAM banks in different indices. Assume a Tile D, with NUM_ROWS = 3, NUM_COLS = 4, MVL = 4 and NUM_BANKS = 4. When NUM_BANKS = 32, the swizzling unit would request data living in different banks but in the same index, guaranteeing an ability to re-order after reading the data out into a crossbar. If we were to read out column 3, we'd like to get (0,3), (1,3) and (2,3) in one wide read from all banks.

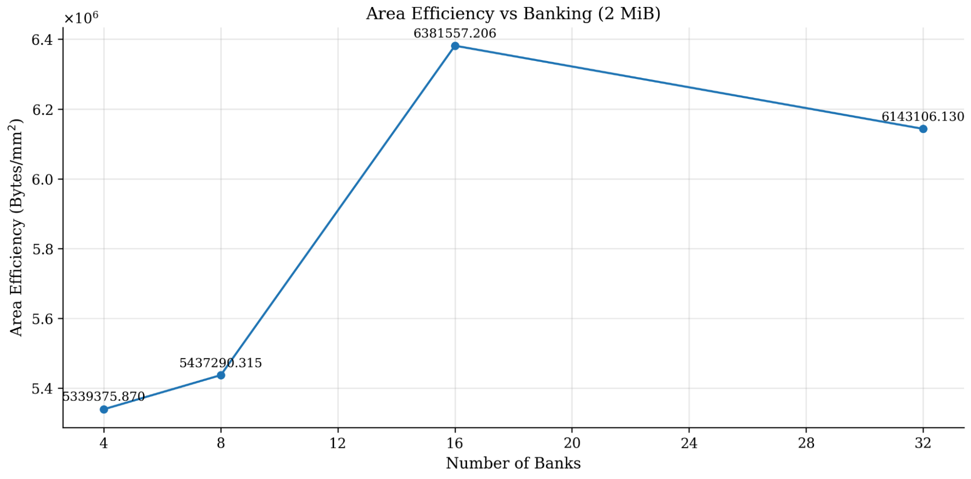


*Figure 7.1.1: SRAM Area Efficiency (Bytes/mm$^2$) Scaling*

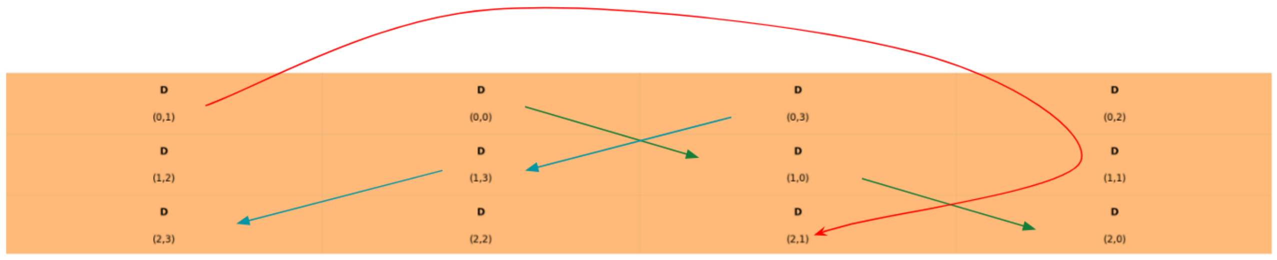


*Figure 7.1.2: Logical view of Swizzled Tiles in SRAM, where MVL = 4*

By changing the number of banks, the design would be required to coalesce multiple (B)FLOAT16 values into one bank, thereby increasing the BANK_WIDTH and port size. While this would benefit area efficiency, as shown in Figure [7.1.1], it would break the guarantees required by swizzling. Taking the example above, if NUM_BANKS = 2, then the first two columns would live in the same bank. This would derive a BANK_WIDTH of four bytes (two values). If we try to read values for column 3 again, (1,3) and (2,3) are now present in different indices in the same bank, leading to a bank conflict.

Thus, the careful designing that was required for swizzling also led us towards an engineering wall where layout can only be optimized by deeply sub-banking. However, this means that all sub-arrays are limited to the same port-size, and would leave more arrays underutilized over the lifetime of a tile. The implications of this are already discussed in Section [6.2]. To overcome this, further iterations of this team will focus on implementing multi-level Scratchpads, where the Level1 instantiation would be few KB in size and will be the only level to support swizzling. Level2 and Level3 versions would disable swizzling, and thus the constraint on the number of banks. Thus, we can optimize the layout of these units by coalescing multiple vector values into a single bank and exploring other knobs for improving area efficiency, delay, and power.

## *Section 7.2: Crossbar Overhead*

As elucidated in Section [3.4], crossbars are a core component of our architecture. However, all explored designs occupy a large portion of the chip area and incur noticeable increments to the total power draw. Using the smallest possible crossbar design – the Benes network – would require a 5KB ROM to meet timing constraints; memory arrays are not trivial in their synthesized area characteristics. As such, we highlight our most impressive feature as subject to trade-offs and intend to optimize other on-chip components to make leeway for it.

# Conclusion

Exploiting regularity of ML workloads to improve compute performance and memory bandwidth utilization is challenging. These programs boil down to a relatively small set of kernels that operate on slices of tensors, called tiles, which exhibit common characteristics that validate investments into specialized accelerator chips. For example, GEMMs are embarrassingly parallelizable, while convolutions can be transformed to mimic the same behaviour. Advancements in the memory subsystem supplying these custom architectures are narrowly explored, while those compute units such as Systolic Arrays are widely researched. Industry giants like Nvidia, AMD, and Qualcomm dominate the AI-focused computing market by focusing on speed-ups in this paradigm.
There is a lack of an end-to-end stack that enables evaluating multiple design choices in accelerator cores because open-source implementations usually focus on modelling the compute units.

To this end, we propose a software-managed memory subsystem, called the Scratchpad, to accelerate the memory transfers for the Atalla Ax01 core. It exploits deterministic memory access patterns and implicit program locality for amortizing overall bandwidth utilization through a custom DMA engine. To avoid bank conflicts in multi-banked SRAM macros, this work evaluates multiple interconnects and macro-layout-strategies. Instead of relying on dynamically learning access patterns to optimize the caching policies, our Scratchpad works with an (in-development) compiler stack to enable fine-grained compile-time scheduling.

All the architectural decisions were evaluated through a Python-based cycle-accurate simulator and emulator. Upon confirming the viability of every choice, the designs were algorithmically parameterized, implemented in SystemVerilog, and testbenched through QuestaSim to acceptable code coverage levels. Post-verification, all Scratchpad sub-modules were synthesized on the MITLL 90nm technology node, and guidelines were defined to help the end-users evaluate tradeoffs between the different parameters. Finally, a programming model was outlined and consequent instructions were created for the programmer to fully utilize the hardware capabilities of the Scratchpad. As such, this work contributes to an open-source framework for concretely evaluating memory subsystems in accelerators architectures. Finally, we hope to further validate our Scratchpad through real-world deployments after integrating it with the other Ax01 subsystems.

## Citations

[1] Y. Zhou et al., "Characterizing and demystifying the Implicit Convolution Algorithm on commercial matrix-multiplication accelerators," 2021 IEEE International Symposium on Workload Characterization (IISWC), pp. 214–225, Nov. 2021. doi:10.1109/iiswc53511.2021.00029

[2] P. Warden, "An engineer's guide to GEMM," Pete Warden's blog, https://petewarden.com/2015/10/25/an-engineers-guide-to-gemm/.

[3] C.-J. Lin, C.-H. Lin, and S.-H. Wang, "Integrated image sensor and light convolutional neural network for Image Classification," Mathematical Problems in Engineering, vol. 2021, pp. 1–7, Mar. 2021. doi:10.1155/2021/5573031

[4] "Tutorial: Opencl SGEMM tuning for Kepler," OpenCL matrix-multiplication SGEMM tutorial, https://cnugteren.github.io/tutorial/pages/page4.html.

[5] S. KPS Mohan et al., "Exploring GEMM and Convolution as a Unified Operation on Atalla's Vector-Core-Driven Systolic Array," 2025

[6] V. Sze Yu-Hsin|Yang, Tien-Ju|Emer, Joel S., Efficient Processing of Deep Neural Networks Sze, Vivienne|Chen, Yu-Hsin|yang, Tien-ju|Emer, Joel S. Springer, 2022.

[7] "Memory Swizzling with Morton ordering," Memory Swizzling with Morton Ordering - Composable Kernel 1.1.0 Documentation, https://rocm.docs.amd.com/projects/composable_kernel/en/latest/conceptual/ck_tile/swizzling_example.html (accessed Dec. 12, 2025).

[8] "What is hip?," What is HIP? - HIP 7.1.52802 Documentation, https://rocm.docs.amd.com/projects/HIP/en/latest/what_is_hip.html (accessed Dec. 12, 2025).

[9] L. Mao, "Cuda Shared Memory Swizzling," Lei Mao's Log Book, https://leimao.github.io/blog/CUDA-Shared-Memory-Swizzling/.

[10] “Nvidia H100 GPU whitepaper,” NVIDIA, https://resources.nvidia.com/en-us-hopper-architecture/nvidia-h100-tensor-c (accessed Dec. 12, 2025).

[11] D. Bernstein, “Verified fast formulas for control bits for permutation networks.” Available: https://eprint.iacr.org/2020/1493.pdf

[12] M.J. Narasimha, “The Batcher-banyan self-routing network: universality and simplification,” *IEEE Transactions on Communications*, vol. 36, no. 10, pp. 1175–1178, Jan. 1988, doi: https://doi.org/10.1109/26.7538.

[13] S. Wei and E. Doug, “An Asynchronous Routing Algorithm for Clos Networks,” *2010 10th International Conference on Application of Concurrency to System Design*, Jun. 2010.

[14] S. Aust and H. Richter, “Real-time Processor Interconnection Network for FPGA-based Multiprocessor System-on-Chip (MPSoC).,” *ResearchGate*, 2010. https://www.researchgate.net/publication/216680094_Real-time_Processor_Interconnection_Network_for_FPGA-based_Multiprocessor_System-on-Chip_MPSoC (accessed Dec. 12, 2025).

[15] L. Portalès, “L1 Triggering on High-Granularity Information at the HL-LHC,” *Instruments*, vol. 6, no. 4, p. 71, Oct. 2022, doi: https://doi.org/10.3390/instruments6040071.

[16] Silvano, “Clos topology for data center networks - part 1,” *Silvano Gai’s Blog*, May 16, 2020. https://silvanogai.github.io/posts/clos-part1/ (accessed Dec. 12, 2025).