Systolic Array-Based Convolution Acceleration for AMP01

Saandiya KPS Mohan*, Akshath Raghav Ravikiran*, Malcolm Lloyd Seibles McClymont*,
Sooraj Chetput Venkataraghavan*, Timothy Francis Hein*
*Elmore Family School of Electrical and Computer Engineering, Purdue University
Email: {mohan76, araviki, mmcclym, schetput, heint}@purdue.edu

***Abstract*—SoCET (system-on-chip extension technologies), features an in-progress Accelerated Matrix Processor (AMP) specifically focused on offloading matrix multiplication tasks from the CPU. Efficient data management is crucial to optimizing AMP performance, particularly in convolutional neural networks (CNNs), where memory bandwidth can become a bottleneck. Traditional approaches to handling convolution operations, such as image-to-column transformation (im2col), require large memory footprints and frequent off-chip memory accesses, leading to increased latency and power consumption. To address this, we explore strategies to optimize data reuse and memory locality by designing a controller that efficiently handles im2col operations, ensuring continuous data supply to the AMP's compute units while minimizing redundant memory operations. Our approach focuses on structuring data flow and storage mechanisms to maximize locality and reduce bandwidth consumption, enabling efficient reuse of overlapping data in convolution operations without unnecessary reloads. This reduces the reliance on expensive DRAM transactions and improves overall hardware utilization. We implement this strategy in SystemVerilog, integrating it with our AMP architecture (systolic array). This work highlights the importance of optimized memory access patterns in hardware accelerators, providing a foundation for existing and future optimizations.**

***Index Terms*—Data Reuse, AI Hardware, Systolic Array, im2col, Locality**

I. Introduction

Convolution is a fundamental operation used in Artificial Intelligence (AI), particularly in signal and image processing, widely adopted in modern deep learning architectures such as Convolutional Neural Networks (CNNs), Generative Adversarial Networks (GANs), and encoder-decoder frameworks. It involves applying a small filter, known as a kernel, across an input to extract local features such as edges, textures, and patterns. By capturing spatial relationships between neighboring pixels or data points, convolution enables neural networks to learn hierarchical feature representations efficiently.

In deep learning, convolutional operations reduce the number of learnable parameters compared to fully connected layers, making models more efficient and less prone to overfitting. CNNs leverage this property to achieve state-of-the-art performance in tasks such as image classification, object detection, and semantic segmentation. Additionally, convolution supports translation invariance, an essential attribute for robust visual recognition systems.

To accelerate these operations, dedicated hardware units such as systolic arrays are often used to perform high-throughput matrix multiplications. However, the memory access patterns involved in convolution pose significant challenges to hardware efficiency. Standard methods like image-to-column (im2col) transformations rearrange input data into a format suitable for general matrix multiplication (GEMM), but at the cost of increased memory usage and redundant off-chip memory accesses [1].

These inefficiencies can severely degrade system performance and energy efficiency, especially in edge devices where memory bandwidth and power are limited. Therefore, optimizing data flow and reuse within hardware accelerators is critical to improving throughput and reducing latency. This paper presents a custom controller design tailored for im2col-based convolution on the Accelerated Matrix Processor (AMP01). The proposed approach focuses on minimizing unnecessary data reloads, maximizing locality, and ensuring continuous data supply to the systolic array compute units, ultimately enhancing hardware utilization and inference efficiency.

II. Background

A. Convolution Fundamentals

Convolution is a mathematical operation that combines two functions to produce a third function, expressing how the shape of one is modified by the other. In the context of deep learning, a discrete 2D convolution is used to process images and extract spatial features. The operation involves sliding a small matrix, called a kernel or filter, over the input data to compute a weighted sum of the values it covers [1].

Mathematically, the 2D convolution between an input matrix I and a kernel K of size k $\times$ k is defined as:

$$O(i,j) = \sum_{m=0}^{k-1} \sum_{n=0}^{k-1} I(i+m, j+n) \cdot K(m,n)$$

where $O(i,j)$ is the output at position (i,j) [10]. This operation is repeated as the kernel moves (or 'slides') across the input, resulting in a feature map that emphasizes specific patterns, such as edges or textures.

B. Key Parameters of Convolution

Several hyperparameters control how the convolution operates and influence the shape and behavior of the resulting feature maps:

- Kernel Size: The dimensions of the filter, typically 3×3, 5×5, etc. Larger kernels can capture broader features but increase computation.
- Stride: The number of pixels that the kernel shifts across the input after each operation. A stride of 1 produces high-resolution output, whereas higher strides downsample the feature map.
- Padding: Adds extra border pixels (often zeros) around the input to control the output size. 'Same' padding keeps the output dimensions close to the input, while 'valid' padding reduces them.
- Dilation: Introduces gaps between kernel elements, allowing the kernel to capture a wider field of view without increasing its size.
- Input channels: For multi-channel inputs (e.g., RGB images), each kernel has weights for each channel, and the output is the sum of convolutions over all channels.
- Number of Filters: Each convolutional layer can apply multiple filters, producing a corresponding number of output channels (or feature maps).

These parameters control both the computational complexity and the representational capacity of the convolutional layers in neural networks.

C. Existing methods

1) **im2col (Image-to-Column)**
   The im2col method restructures a multidimensional input feature map into a 2D matrix, where each column corresponds to a flattened patch (receptive field) of the input that the kernel slides over. This allows the convolution operation to be expressed as a standard matrix multiplication between the im2col matrix and a reshaped kernel matrix. [1]
   Although this enables efficient execution using BLAS-like libraries, it introduces several drawbacks:
   - Excessive Memory Usage: Overlapping receptive fields lead to significant data duplication. For small strides or large inputs, the im2col matrix can be orders of magnitude larger than the original input.
   - Bandwidth Overhead: The large intermediate matrix often exceeds on-chip memory, requiring frequent access to off-chip DRAM.
   - Loss of Data Locality: Once flattened, the pixels are in non-adjacent memory addresses, which undermines temporal and spatial reuse in hardware pipelines.
2) **im2row (Image-to-Row)**
   Similarly, the im2row technique flattens patches into rows instead of columns. It is commonly used in CPU-optimized libraries and may offer better cache alignment in certain architectures. However, the fundamental drawbacks remain the same—data redundancy, large memory footprint, and lack of alignment with hardware-specific dataflows. [1]
3) **Winograd Convolution**
   Winograd's minimal filtering algorithm is an optimization technique that reduces the number of multiplications required in small convolutional kernels, particularly for 3×3 filters [2]. Instead of directly performing the convolution in the spatial domain, the Winograd algorithm transforms both the input tile and the kernel into another domain where element-wise multiplication is performed. After the multiplications, the result is transformed back to the spatial domain [7].
   For example, in the commonly used $F(2 \times 2, 3 \times 3)$ Winograd configuration, a 4×4 input tile and a 3×3 kernel produce a 2×2 output tile. The computation follows this structure:

$$Y = A^T \left[(GgG^T) \odot (BTdB^T) \right] A$$

   Where:
   - g is the 3×3 kernel
   - d is the 4×4 input tile
   - G, B, and A are transform matrices specific to the Winograd configuration
   - $\odot$ denotes element-wise (Hadamard) multiplication

   This transformation significantly reduces the number of multiplications (from 9 to 4 in the $F(2 \times 2, 3 \times 3)$ case), improving arithmetic intensity [8]. However, it introduces additional additions, and the fixed transform matrices must be carefully managed [5].

III. Constraints

Designing a hardware-efficient convolution engine for the Accelerated Matrix Processor (AMP01) comes with a unique set of constraints, both architectural and functional. While convolution operations can be mapped to matrix multiplication through software transformations like im2col or im2row, these approaches are fundamentally mismatched with the AMP's design philosophy and existing infrastructure.

A. Incompatibility of im2col/im2row with AMP

The im2col and im2row methods restructure input feature maps into flattened matrices to facilitate general matrix multiplication. However, AMP is built around a systolic array architecture, optimized for tightly controlled, streaming dataflows and minimal memory redundancy [3]. Integrating im2col or im2row into AMP introduces several critical problems:

- Data Duplication: im2col inherently duplicates overlapping regions of the input, inflating memory requirements. AMP's limited on-chip buffer cannot support this overhead.
- Non-Streaming Data Access: These transformations require loading large flattened matrices into memory before computation. This clashes with AMP's expectation of continuous, small-batch data streaming into the systolic array.
- Loss of Locality: Flattened data discards spatial relationships that are essential for buffer reuse and efficient pipeline operation within AMP.

- Control Complexity: Supporting im2col would require complex control logic for data unpacking and reshaping, increasing the latency and hardware footprint, undermining the lightweight, low-latency design goals of AMP.

B. Hardware and Architectural Constraints

In addition to incompatibility with im2col-style dataflows, this project is also constrained by the existing design and capabilities of the AMP00:

- Pre-Defined Systolic Array: The AMP already includes a fixed systolic array optimized for matrix multiplications. Modifying or redesigning this array is outside the project scope, which means all new memory management strategies must interface with the current hardware as-is.
- Fixed Compute Unit Scheduling: The flow of operands through the array is predetermined, so input buffers must align with these patterns without stalling the compute pipeline.
- Limited On-Chip Memory: AMP's buffer space is finite, and any data reuse strategy must be carefully optimized to fit within strict resource budgets. This requires efficient tiling and overlap-aware reuse mechanisms.
- Latency and Throughput Goals: The system must support real-time or near-real-time inference workloads. This prohibits designs that introduce significant pre-processing overheads, like full im2col transformations.

Given these constraints, the challenge is to develop a lightweight, high-reuse convolution controller design that can efficiently feed overlapping windows of convolutional data directly into the systolic array without transformation, duplication, or external storage dependency. The next section introduces such a buffer architecture tailored for this environment.

C. Systolic Array Overview

The systolic array is a highly parallel hardware architecture designed for efficient implementation of dense linear algebra operations, particularly matrix multiplication. It consists of a grid of Processing Elements (PEs) that perform simple arithmetic operations (typically multiply-accumulate, or MAC) in a pipelined and synchronized fashion. Each PE passes data to its neighboring PEs in a fixed pattern, creating a rhythmic, wave-like flow of data through the array.

In the systolic array used in Accelerated Matrix Processor (AMP00), PEs are arranged in a two-dimensional grid. Input data flows through the array in two directions, which are rows, one operand matrix (typically the input activation or intermediate feature tile) is fed row-by-row across the array and columns, the other operand matrix (typically the weight matrix or kernel tile) is streamed down the columns.

Each PE receives an element from the row input and an element from the column input, performs a multiplication, adds the result to a running partial sum, and then passes the inputs to the next PE along its row and column. This results in each PE contributing to the computation of a single element in the output matrix [10].

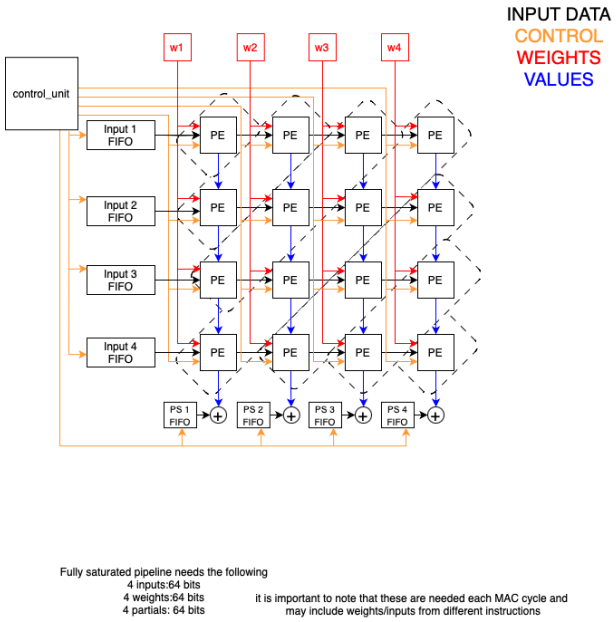

Fig. 1. Systolic array block diagram.

The systolic array's key advantage lies in its predictable and regular data movement, which reduces control complexity and enables high throughput. It also minimizes off-chip memory accesses by reusing data within the array as it flows through.

In the AMP, the systolic array is designed to handle general matrix multiplications (GEMM), which are a core component of many machine learning workloads. The grid size is 4×4, and its performance is highly dependent on receiving a steady stream of data from both operands. This streaming model allows the AMP to achieve high utilization of its compute units, provided the data is prepared and delivered in an efficient manner [10].

However, while the systolic array excels at matrix multiplication, it does not directly support convolution operations, which involve sliding window patterns, data reuse, and non-uniform access patterns. To bridge this gap without modifying the systolic array itself, external logic is required to orchestrate data movement in a way that maps convolutions into compatible matrix multiplications. This is the responsibility of the convolution controller discussed in the next section.

IV. Design Choices Justification

Throughout the design process, we evaluated multiple convolution strategies to determine the most hardware-efficient and compatible method for our AMP. Initially, Winograd convolution appeared to be a promising candidate due to its ability to reduce the number of multiplications required for small convolutions [2]. However, after deeper research and attempts to map Winograd onto hardware, we encountered significant challenges. The algorithm requires multiple transformation matrices that vary with kernel and tile size, and these matrices must be applied to both the input data and the kernel before and after element-wise multiplication. Generating or storing

these matrices on-chip would consume considerable logic and memory resources. Moreover, integrating the necessary pre and post-processing steps introduced irregular data access patterns that conflicted with the regular, streaming nature of our AMP's systolic array. These observations led us to conclude that while Winograd is mathematically efficient, its control complexity and dataflow misalignment make it unsuitable for our constrained hardware architecture.

In parallel, we explored data restructuring techniques like im2row and im2col, both of which convert convolution into matrix multiplication by reshaping the input image into a series of overlapping patches [4]. After testing both layouts, we found that im2col was a better fit for our AMP. The im2col method flattens each receptive field (e.g., 3×3 patch) into a column vector, which aligns cleanly with the way our systolic array consumes input matrices—row-by-row for one operand and column-by-column for the other. In contrast, im2row flattens patches into row vectors, which would require transposing data before feeding it into the systolic array, complicating the controller and reducing throughput.

Other than that, we also experimented with a buffer-based design. This method involved holding patches of the input matrix in four separate buffers, each preloaded with values corresponding to a region of the input [4]. The controller would then read from these buffers and map the values to the correct inputs of the systolic array. While this approach worked in simulation, it introduced significant hardware overhead due to the need to manage multiple read/write pointers, and handle boundary conditions. The buffers also added unnecessary storage redundancy, as many values were reused across multiple tiles due to convolution's overlapping nature. After further architectural exploration, we discovered that by carefully structuring data access patterns and managing the systolic array input FIFOs intelligently, we could eliminate the need for preloaded buffers entirely. The scratchpad was redesigned to provide burst access to $k \times k$ tiles directly, and the controller was modified to push those values in real time into the appropriate FIFOs. This realization dramatically simplified our design and reduced both logic and memory resource usage, while maintaining performance. It also reinforced our emphasis on dataflow-aware design, where efficient movement and reuse of data took priority over brute-force buffering.

Through this process, we learned that algorithm-hardware alignment is just as critical as algorithmic efficiency. The elegance of Winograd did not translate into practical performance gains due to architectural mismatches. On the other hand, im2col offered a more straightforward and scalable solution that required minimal adaptation of our existing AMP. This realization ultimately shaped the direction of our convolution controller design.

V. Final Designs

Considering the architectural constraints and trade-offs discussed in earlier sections, we developed a convolution controller integrated into the AMP01 system. The design consists of four core components: a modified systolic array optimized for column-wise data flow, a scratchpad memory system redesigned for banked SRAM access, a custom convolution controller responsible for data sequencing, and the AMP01 top-level integration which orchestrates the interaction between all components. These combined components work together to efficiently execute convolution operations using im2col while maximizing hardware reuse from our original matrix multiplication design.

A. AMP01

AMP01 is the top-level integration of our Accelerated Matrix Processor (AMP). It brings together the compute, memory, and control subsystems needed to perform matrix operations independently of the core CPU. AMP01 consists of the systolic array, scratchpad memory, convolution controller, and interfaces to off-chip memory. Within this layout, the convolution controller sits between the CPU-side command input and the scratchpad, issuing instructions to control memory access. The scratchpad feeds processed data to the systolic array, which performs matrix multiplications on im2col-reformatted data. The partial sum register collects and filters outputs, selectively storing only meaningful results. AMP01 is modular, allowing future extensions such as ReLU activations, pooling, or batch normalization modules to be inserted between the compute and memory stages. This modularity ensures flexibility without compromising the lightweight footprint of the design.

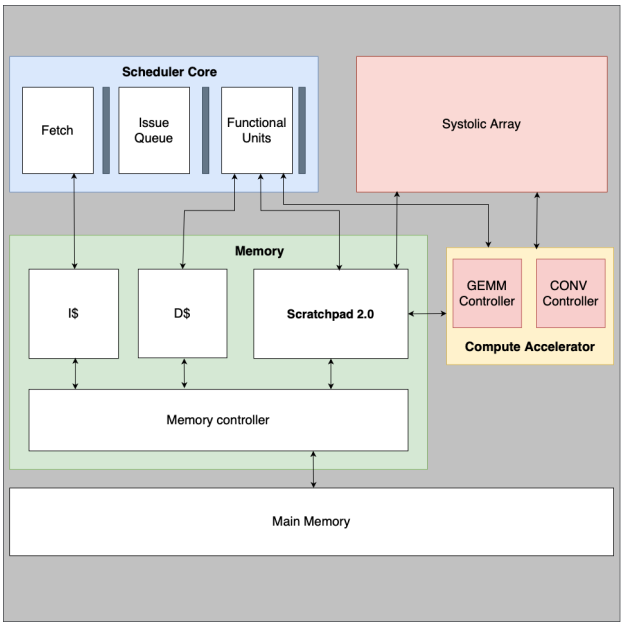

Fig. 2. Top Level RTL of AMP01.

B. Systolic Array

Our AMP01 includes a 32×32 systolic array composed of FP16 multiply-accumulate (MAC) units arranged in a regular grid. The original design supported row-wise data streaming, where rows of an input matrix would be pushed across the horizontal direction and the corresponding kernel

weights are stationary where outputs then flow vertically. This format is ideal for general matrix multiplication but conflicts with im2col's layout, where input patches are flattened into columns.

To resolve this mismatch, we reoriented the dataflow in the systolic array to support column-by-column loading. According to systolic array in Figure 3, flattened k $\times$ k convolution tiles from the input feature map (prepared via im2col) are streamed vertically into the array. This reconfiguration required changes to the input FIFO management, ensuring the correct ordering and alignment of data along the vertical axis. Each MAC unit receives input from the FIFO above and a delayed weight from its left neighbor, maintaining the original systolic processing pattern.

The array remains fully pipelined, with no changes to the core computation, allowing us to reuse the optimized design. This approach preserves throughput while aligning with the new memory and controller interfaces.

C. Scratchpad

The original AMP00 prototype used a register-based scratchpad, suitable for manually managed data movement but inefficient for dynamic convolution operations. We redesigned the scratchpad as a linearly addressed, banked SRAM system with 32 independent memory banks. Each bank stores a segment of the input image, weights, or partial sums. To allow parallel access to multiple FP16 values from a tile, we implemented swizzling, a data-layout transformation that ensures adjacent rows or columns are distributed across different banks to avoid access conflicts.

This scratchpad architecture allows the convolution controller to perform high-speed burst reads of im2col tiles from memory. When a k $\times$ k tile needs to be loaded, the controller specifies the base address, and the scratchpad returns all k^2 values in a single operation. This supports both horizontal and vertical data access patterns, essential for extracting patches during stride-based movement. The scratchpad also differentiates between GEMM and convolution operations using a GEMM controller and convolution controller in the compute accelerator.

The memory interface is synchronized with the controller FSM and supports backpressure and flow control signals, ensuring data integrity. This redesign, implemented by Akshath, ensures the system scales with larger feature maps and kernel sizes while keeping on-chip memory access latency low.

D. Convolution Controller

The convolution controller is the key innovation in our system. It is responsible for converting high-level convolution instructions (e.g., kernel size, stride, padding, and input dimensions) into precise, low-level memory access patterns and dataflow control signals for the scratchpad and systolic array.

At the heart of the controller is a finite state machine (FSM) that iterates through the input image using nested loops over rows and columns. For each position, it computes the base

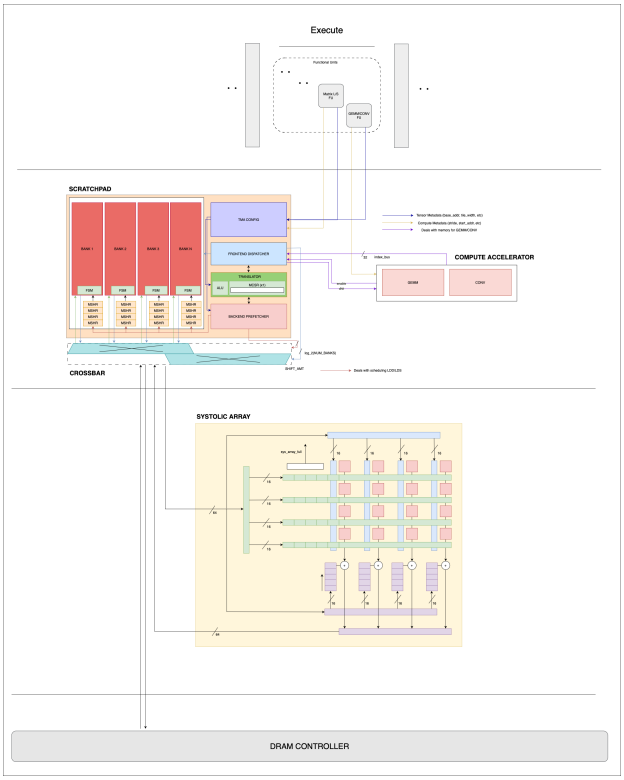

Fig. 3. Scratchpad block diagram.

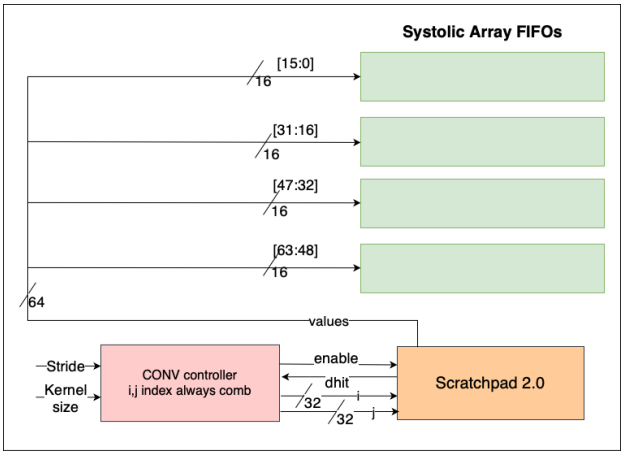

Fig. 4. Convolution controller block diagram.

address of the corresponding k $\times$ k tile in the scratchpad using the formula:

$$\text{Address}(i, j) = \text{Base} + i \times \text{RowStride} + j$$

where i, j denote the tile's top-left corner, and strides depend on the user-specified stride parameter. The controller sequentially instructs the scratchpad to read from these locations and flatten the result into a vector (column).

Once the tile is received, the controller distributes the FP16 values to the appropriate input FIFOs of the systolic array. The mapping logic ensures spatial alignment between the data and

compute units. For a 3 × 3 kernel, this involves broadcasting 9 values across columns such that each column of the systolic array receives one value per cycle for 9 cycles.

For example, for a 4 × 4 kernel, stride = 1, and a 6 × 6 input matrix, the weights will be loaded in order in the PEs of systolic array. As for the inputs, refer to the Table 1. The values highlighted in yellow represent the first input tile extracted from the 4 × 4 convolution kernel, corresponding to the initial convolution window over the input feature map.

Example of 6 × 6 input matrix:

$$\begin{bmatrix} a_{00} & a_{01} & a_{02} & a_{03} & a_{04} & a_{05} \\ a_{10} & a_{11} & a_{12} & a_{13} & a_{14} & a_{15} \\ a_{20} & a_{21} & a_{22} & a_{23} & a_{24} & a_{25} \\ a_{30} & a_{31} & a_{32} & a_{33} & a_{34} & a_{35} \\ a_{40} & a_{41} & a_{42} & a_{43} & a_{44} & a_{45} \\ a_{50} & a_{51} & a_{52} & a_{53} & a_{54} & a_{55} \end{bmatrix}$$

TABLE I
EXAMPLE OF INPUT DATA ARRANGEMENT IN SYSTOLIC ARRAY

$a21$	$a11$	$a01$	$a30$	$a20$	$a10$	$a00$
$a12$	$a02$	$a31$	$a21$	$a11$	$a01$	0
$a03$	$a32$	$a22$	$a12$	$a02$	0	0
$a33$	$a23$	$a13$	$a03$	0	0	0

$\longrightarrow$ To systolic array.

Importantly, the systolic array computes independently of the FSM. Once loaded, it produces a matrix output corresponding to one or more output channels. The controller uses masking logic to filter invalid partial sums, particularly near the image edges or padded regions. Only valid outputs are collected in the partial sum register, reducing post-processing overhead.

The FSM is parameterized to handle various kernel sizes and strides, allowing the controller to generalize across CNN layers. The modular structure of this controller enables easy testing, reconfiguration, and potential extensions to support features like dilated convolution or grouped convolution in future designs.

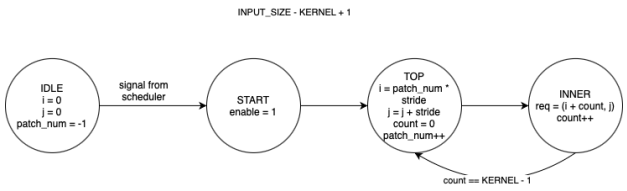

Fig. 5. Convolution controller FSM.

To validate the correctness and performance of our design before RTL implementation, we developed a full Python simulator that models the behavior of the convolution controller, and the systolic array. This simulator helped us test various convolution parameters, detect edge cases (like stride-induced overlap), and confirm the correct mapping of partial sums to output features. It also enabled cycle-accurate visualization of data movement, helping debug timing and synchronization issues before moving to SystemVerilog. The simulator proved essential in refining our control logic and gave us high confidence that the hardware design would behave as expected under real-world CNN workloads.

VI. ADVANTAGES OF CONVOLUTION CONTROLLER

The convolution controller architecture designed for AMP01 offers several hardware-level benefits over traditional software-based or buffer-heavy implementations:

- **Reduced Memory Footprint:** By avoiding intermediate buffer storage and reusing overlapping input data directly from the scratchpad, our approach minimizes the memory required for convolution operations. This leads to a leaner hardware design that better fits within constrained on-chip resources.
- **Fewer DRAM Accesses:** Conventional software-based im2col methods often involve repeated DRAM reads for overlapping input regions. Our controller ensures that input values are fetched once from off-chip memory and then reused efficiently through the scratchpad and systolic array, greatly reducing expensive and power-hungry DRAM transactions.
- **Optimized Data Reuse:** The integration of swizzling in the scratchpad and precise data routing by the convolution controller enables high data locality, further enhancing performance by avoiding duplicated data movements.

Overall, these advantages contribute to lower power consumption, faster convolution execution, and better utilization of the AMP01 hardware.

FUTURE PLANS

The next steps for this project involve the full hardware realization and integration of the proposed design. First, we will implement the convolution controller's finite state machine (FSM) and supporting logic in SystemVerilog, along with a comprehensive testbench to verify its correctness, timing, and dataflow behavior under various convolution configurations.

After successful standalone validation, we plan to integrate the convolution controller with other subsystems of AMP01, including the modified scratchpad memory and column-wise systolic array. This will involve ensuring proper handshake protocols, synchronization, and performance tuning across the AMP pipeline.

These efforts aim to move the convolution controller from conceptual simulation into a functional, verified RTL module, paving the way for real-world convolution acceleration on AMP01.

CONCLUSION

In this project, we developed a convolution controller integrated into Accelerated Matrix Processor (AMP01) to efficiently handle convolutional workloads on hardware. Starting from an evaluation of multiple convolution strategies including Winograd, buffer-based methods, and im2row, we identified im2col as the most compatible approach given the constraints

of our existing architecture. Our design strategically avoids unnecessary buffers and instead optimizes data routing and reuse by working in tandem with a modified systolic array and scratchpad memory.

Through this process, we gained a deeper understanding of the trade-offs between mathematical efficiency and hardware feasibility, particularly with the complexity of implementing Winograd transformations on constrained compute units. The decision to switch from a row-based to a column-based systolic array, coupled with a redesigned scratchpad using swizzling, allowed us to fully utilize the existing matrix multiplication infrastructure in a resource-efficient way.

The convolution controller FSM, validated through a comprehensive Python simulator, successfully manages data movement, ensures synchronization across hardware components, and maintains low memory overhead while delivering accurate output activations. This work underscores the importance of data locality, reuse, and hardware-software co-design in building efficient CNN accelerators. Our architecture now provides a solid foundation for future expansions, including support for varying kernel sizes, dynamic stride handling, and pipelined multi-layer inference.

ACKNOWLEDGMENT

The authors would like to thank the System-on-Chip Extension Technologies (SoCET) group, part of the Vertically Integrated Projects (VIP) program at Purdue University, for providing the hardware platform, development environment, and technical mentorship that enabled this work. We are especially grateful to Dr. Anand Raghunathan and Dr. Mark Johnson for their expert guidance, encouragement, and invaluable insights throughout the project. Their support was instrumental in the design and integration of the convolution controller into the AMP01 processor.

REFERENCES

[1] B. QoChuk, "im2col Convolution," OpenGenus IQ: Learn Algorithms, DL, System Design, 2025. https://iq.opengenus.org/im2col/

[2] R. Andri, B. Bussolino, A. Cipolletta, L. Cavigelli, and Z. Wang, "Going Further With Winograd Convolutions: Tap-Wise Quantization for Efficient Inference on 4x4 Tile," arXiv.org. https://doi.org/10.48550/arXiv.2209.12982

[3] N. P. Jouppi et al., "In-Datacenter Performance Analysis of a Tensor Processing Unit," arXiv.org. https://doi.org/10.48550/arXiv.1704.04760

[4] P. Wang et al., "An Efficient im2row-Based Fast Convolution Algorithm for ARM Cortex-M MCUs," IEEE Access, vol. 9, no. 1, pp. 124384–124395, 2021, doi: 10.1109/access.2021.3110827.

[5] D. Xie, Z. Jia, Z. Zhang, and X. Jin, "Optimizing half precision Winograd convolution on ARM many-core processors," in Proceedings of the 13th ACM SIGOPS Asia-Pacific Workshop on Systems, New York, NY, USA: ACM, Aug. 2022, pp. 53–60. https://doi.org/10.1145/3546591.3547529

[6] H. Genc et al., "Gemmini: Enabling Systematic Deep-Learning Architecture Evaluation via Full-Stack Integration," arXiv.org. https://arxiv.org/abs/1911.09925

[7] X. Liu, J. Pool, S. Han, and W. J. Dally, "Efficient Sparse-Winograd Convolutional Neural Networks," arXiv.org. https://arxiv.org/abs/1802.06367

[8] A. Lavin and S. Gray, "Fast Algorithms for Convolutional Neural Networks," arXiv.org. https://arxiv.org/abs/1509.09308

[9] H Logix & Solutions, "A Simple 2D Convolution Using Systolic Arrays." Sep. 15, 2020. Accessed: May 01, 2025. https://youtu.be/WYw9guur0Lo?si=HMWOybgXEBVAgWRH

[10] T. Raja, "Systolic Array Data Flows for Efficient Matrix Multiplication in Deep Neural Networks," arXiv.org. https://doi.org/10.48550/arXiv.2410.22595