FPGA-Based Systolic Array Accelerator for GeMM Operations

Cameron Barbour
cbarbr@umich.edu
University of Michigan
Ann Arbor, Michigan, USA

Andrew Keil
drewkeil@umich.edu
University of Michigan
Ann Arbor, Michigan, USA

Tong Sing Wu
tongsing@umich.edu
University of Michigan
Ann Arbor, Michigan, USA

Guanghao Chen
guanghao@umich.edu
University of Michigan
Ann Arbor, Michigan, USA

Maatla Sefawe
maatla@umich.edu
University of Michigan
Ann Arbor, Michigan, USA

Abstract

Neural Networks make extensive use of matrix operations, most commonly matrix matrix multiplication (GeMM). Convolutional Neural Networks, which are common in edge devices used for computer vision, can also be made to use GeMM through convolution lowering. These workloads are highly parallel and are often run on GPUs to exploit this parallelism. While GPUs provide high throughput, they require significant power and are not always ideal for embedded or edge deployments. A dedicated accelerator for GeMM operations can exploit this parallelism with a significantly lower energy cost by relying on high data reuse and mostly local data movement. In this paper we implemented a weight stationary systolic-array based GeMM accelerator designed to run on an FPGA. Our design is able to run at 100 MHz clock frequency with the PE array itself using only ≈1.4% of the available LUTs on the device. We were able to use our design to successfully run a simple network trained on the MNIST dataset.

1 Introduction

Convolutional neural networks (CNNs) have become the dominant model for a wide range of perception tasks, including image classification, object detection, and semantic segmentation [8, 9]. As models grow deeper and wider [5], and as applications migrate from cloud servers to edge platforms, the computational and memory demands of CNN inference increasingly strain conventional processors. A single modern CNN can require billions of multiply–accumulate (MAC) operations per input and must often run under tight latency and power constraints on embedded devices [10]. General-purpose CPUs and even GPUs are not always well matched to these workloads, motivating a large body of work on specialized CNN accelerators that exploit data reuse, parallelism, and domain-specific optimizations to improve energy efficiency and throughput [1, 6].

Within this landscape, systolic arrays have emerged as a compelling architectural template for dense linear-algebra workloads [9]. By organizing processing elements (PEs) in a regular two-dimensional grid and streaming operands across the array, systolic architectures can sustain high utilization with local communication and predictable data movement. Prior work has shown that mapping convolution to matrix multiplication [3, 6] enables CNN layers to be efficiently executed on systolic arrays. Furthermore a variety of ASIC and FPGA designs have leveraged weight-stationary, output-stationary, or row-stationary dataflows to maximize on-chip reuse of activations and weights[1, 7]. Despite these advances, designing efficient systolic accelerators for FPGAs remains challenging; architects must rigorously balance compute density, on-chip memory capacity, and off-chip bandwidth, particularly given the scarce DSP blocks and routing resources available on edge devices [11].

To address these challenges, this work presents a custom CNN accelerator implemented on an FPGA that prioritizes efficient convolution through a systolic array architecture. We design and implement a parameterizable 2D systolic array utilizing a weight-stationary strategy [11]. In this scheme, filter weights are preloaded and retained within the PEs while input activations stream through the array. This organization maximizes the reuse of weights which is particularly beneficial for standard small kernels.

In this stage of the work, we focus specifically on the compute efficiency of the systolic core. We assume that input activations and weights are pre-formatted into the necessary memory layout (e.g., via an offline im2col transformation) prior to execution. This design choice allows us to isolate the performance characteristics of the systolic array and on-chip buffering hierarchy without the complexity of on-the-fly data transformation logic.

In summary, this work makes the following contributions:

- Architecture Design: We design and implement a parameterizable, weight-stationary systolic array on an FPGA, complete with a supporting control and buffering microarchitecture for pre-formatted data streams.
- Experimental Evaluation: We characterize the accelerator's performance, resource utilization, and scalability across varying array sizes.
- Bottleneck Analysis: We identify critical system limitations and discuss the necessity of architectural extensions.

2 Architecture Overview

The accelerator is made up of an array of processing elements, a controller, BRAMS storing weights, input activations, and psums, and the host interface. It was designed to be scalable using parameterization. The 3x3 PE configuration shown below requires 9 BRAMS. Therefore, the host interface and memory map also scales with the PE array configuration. The controller must also be able to account for the time taken to load weights into the PE array and perform the GEMM by streaming input activations down the array with PSUMs streaming into the PSUM BRAMS.

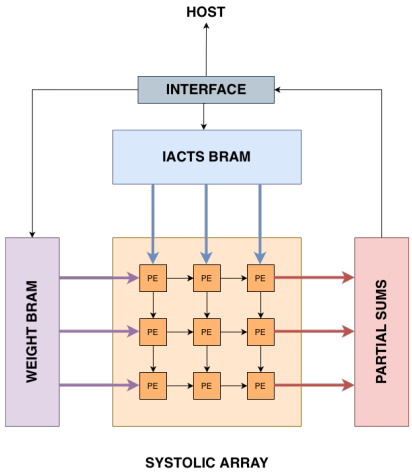

Figure 1: Accelerator Architecture.

The architecture of each processing element (PE) is relatively simple. Because this is a weight-stationary systolic array, weights must first be loaded into the PE array. This is accomplished using the weight buffer and global load weight signal. Load weight is asserted high until the final column of weights is loaded. At that point, the weights for each PE are now stored in the array. We recognize that a global load weight signal may not meet timing when the PE array scales much larger. The signal could quite easily be pipelined or split across the fabric as necessary. Following the loading of weights into the PE array, we now stream input activations top down into the PE array. Each PE performs and multiply with the weight and accumulate with the partial sum of the left neighboring PE's partial sum before passing the partial sum to the right PE. The incoming input activation is also registered and passed onto the lower PE. The multiply and accumulate are performed within a single clock cycle. If running with a higher speed clock, this operation may need to be pipelined. This additional delay can easily be accounted for by the controller.

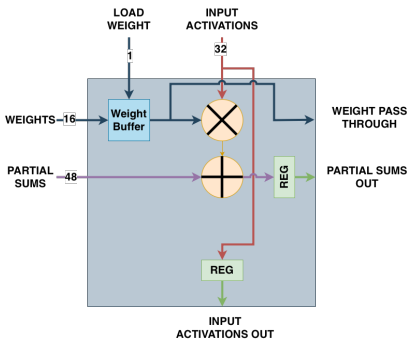

Figure 2: PE Architecture.

The controller runs off a 4 state FSM with counters for the load weight cycle and the compute cycle. A go signal from the controller is sampled for its rising edge to kick off the GEMM moving the FSM from IDLE to LOAD_WEIGHTS. While in the LOAD_WEIGHTS state, the global load weight signal is asserted to all PEs to store weights in their weight buffer. At the same time, a counter runs to track how long weights have been loaded. The counter is compared to a parameterizable constant to change the number of cycles for loading depending on the size of the array. In general, the number of load weight cycles is equal to the number of columns in the array. Following the loading of weights, the FSM moves to the COMPUTE state where input activations are streamed into the PE array from BRAM and PSUMs are written to the BRAMS when valid. Due to the staggered nature streaming input activations into the PE array, each column loading input activations has a separate load signal generated using a for loop. A similar implementation based on the number of rows is used for validating partial sums being put into BRAM. A counter for compute cycles is compared to a parameterizable constant to change the number of compute cycles depending on the size of array while in the COMPUTE state similar to the LOAD_WEIGHTS state. When compute is done, the FSM moves to the DONE state and asserts the done signal indicating to the host interface that the GEMM operation is complete and PSUMS can be read out of the BRAMS. The FSM does not return to the IDLE state. It will sit in DONE until another rising edge of GO is detected.

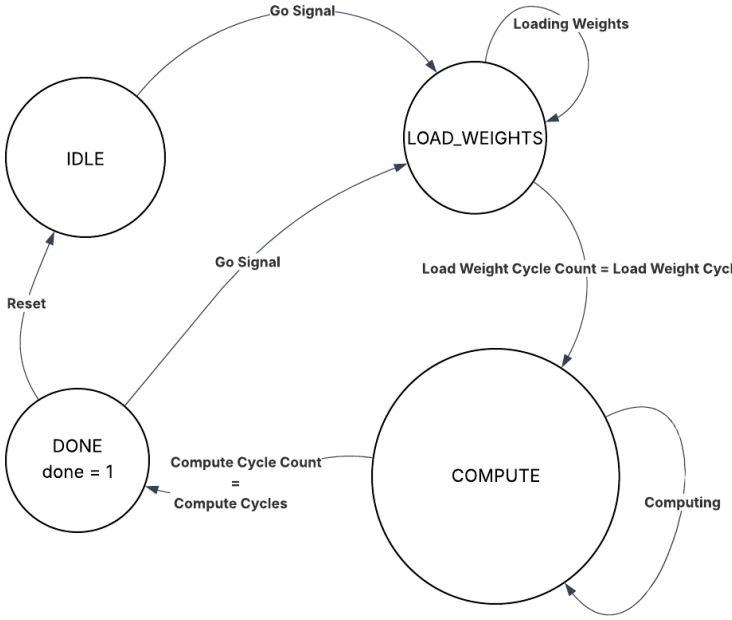

Figure 3: Controller FSM.

The host interface is implemented in a Vivado block design and makes use of AXI infrastructure IP. A Clocking Wizard IP is used to synchronize an internal 100MHz clock to the external 100MHz board clocks. The core provides a lock signal fed into a processor system reset which holds AXI SmartConnect and peripherals in reset while the clock is not locked. There are two AXI masters in the host interface. The JTAG to AXI master IP core allows for JTAG

communication to the accelerator. This was useful for early debug and communication to the design using Vivado Lab Manager. The UART to AXI master was added to allow for a simple interface to a laptop for proof of concept and python integration using the PySerial package. The two AXI masters are fed into an AXI SmartConnect controlling 10 AXI slaves. 9 AXI slaves are BRAMs for weights, input activations, and psums. The number of AXI slaves would scale as the PE array grows. Multiple SmartConnects can be chained together in this case to bypass the 16 slave limit. The remaining 1 AXI slave is dedicated to the AXI GPIO for controlling the go signal and reading the done signal. 1 bit of the GPIO_DATA register is written to assert the go signal after loading weights and input activations into the BRAMS using 6 AXI BRAM controllers. The GPIO_TRI register bit must be 0 to be configured at output and 1 when configured as input. When configured as input it is read to check for the done signal from the controller.

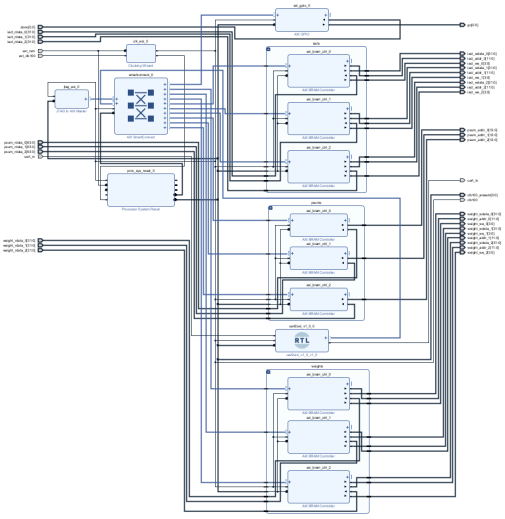

Figure 4: Host Interface Architecture.

The address map of the accelerator allocates at minimum 4KB of memory for each weight row and input activation column. The design uses 16 bit weights and 32 bits input activations each stored in 4 byte words meaning we can store up to 1K weight and input activations in each BRAM. This should allow for a 1K by 1K PE array. Because the resulting PSUMS are 48 bits, 8KB is allocated for each PSUM ROW to allow for 1K PSUMS in each BRAM to be stored. The AXI GPIO is given 128 bytes, the minimum allowed by Vivado block designs. The weights and input activation BRAMS are write only and the PSUM BRAMS are read only. This was achieved by tieing down necessary native BRAM interface signals.

Name	Interface	Slave Segment	Master Base Addr...	Range	Master High Address
Network 0 (/jtag_axi_0/Data, /uart2axi_v1_0_0/m00_axi)					
/jtag_axi_0					
/jtag_axi_0/Data (32 address bits : 4G)					
/axi_gpio_0/S_AXI	S_AXI	Reg	0x0	128	0x7F
/weights/axi_bram_ctrl_0/S_AXI	S_AXI	Mem0	0x1000	4K	0x1FFF
/weights/axi_bram_ctrl_1/S_AXI	S_AXI	Mem0	0x2000	4K	0x2FFF
/weights/axi_bram_ctrl_2/S_AXI	S_AXI	Mem0	0x3000	4K	0x3FFF
/iacts/axi_bram_ctrl_0/S_AXI	S_AXI	Mem0	0x4000	4K	0x4FFF
/iacts/axi_bram_ctrl_1/S_AXI	S_AXI	Mem0	0x5000	4K	0x5FFF
/iacts/axi_bram_ctrl_2/S_AXI	S_AXI	Mem0	0x6000	4K	0x6FFF
/psums/axi_bram_ctrl_0/S_AXI	S_AXI	Mem0	0x8000	8K	0x9FFF
/psums/axi_bram_ctrl_1/S_AXI	S_AXI	Mem0	0xA000	8K	0xBFFF
/psums/axi_bram_ctrl_2/S_AXI	S_AXI	Mem0	0xC000	8K	0xDFFF
/uart2axi_v1_0_0					
/uart2axi_v1_0_0/m00_axi (32 address bits : 4G)					
/axi_gpio_0/S_AXI	S_AXI	Reg	0x0	128	0x7F
/weights/axi_bram_ctrl_0/S_AXI	S_AXI	Mem0	0x1000	4K	0x1FFF
/weights/axi_bram_ctrl_1/S_AXI	S_AXI	Mem0	0x2000	4K	0x2FFF
/weights/axi_bram_ctrl_2/S_AXI	S_AXI	Mem0	0x3000	4K	0x3FFF
/iacts/axi_bram_ctrl_0/S_AXI	S_AXI	Mem0	0x4000	4K	0x4FFF
/iacts/axi_bram_ctrl_1/S_AXI	S_AXI	Mem0	0x5000	4K	0x5FFF
/iacts/axi_bram_ctrl_2/S_AXI	S_AXI	Mem0	0x6000	4K	0x6FFF
/psums/axi_bram_ctrl_0/S_AXI	S_AXI	Mem0	0x8000	8K	0x9FFF
/psums/axi_bram_ctrl_1/S_AXI	S_AXI	Mem0	0xA000	8K	0xBFFF
/psums/axi_bram_ctrl_2/S_AXI	S_AXI	Mem0	0xC000	8K	0xDFFF

Figure 5: Host Interface Address Map.

3 Methodology

The systolic array, memory controller FSM, and UART-to-AXI interface bridge are implemented entirely in SystemVerilog RTL. To integrate the accelerator into a full system, the AXI interconnect and BRAM controllers are created in the Vivado block design environment. Tiled weights, input activations, and output partial sums are stored in BRAM, while the AXI-GPIO registers provide the accelerator's control interface, including start, done, and matrix dimension configuration. The complete design is synthesized using the standard Xilinx Vivado tool flow before being deployed to the FPGA to validate the data flow with the host PC. We prototyped our design on the Xilinx Artix-7 and Xilinx Zynq-7000 FPGA

To demonstrate the accelerator, we created a Python GUI application with a custom Device API built on PySerial to initiate operations on the FPGA for the MNIST dataset. The model consists of a single dense layer mapping 784 features to 10 output classes (representing numbers 0-9). We chose a single dense FC layer because the workload is exactly a GeMM operation of a 10x784 weight matrix and a 784x1 input activation matrix (the 28x28 image). Specifically, the output matrix, which is the product of the input activation matrix and the weight matrix, will return a 1x10 matrix where the index with the greatest element corresponds to the model's prediction. While it is technically possible to extend the classifier to multiple fully connected layers, such as transitioning from 784 to 128 to 10, this would require buffering multiple layers in the hardware for optimal processing, which increases the design complexity. Doing multiple fully-connected layers would require a input vector (1x784) * weight matrix (784*128) = 1x128 output matrix -> Relu -> output matrix of layer 1 (1x128) * weight matrix (128x10) = output matrix (1x10). Due to the size of the MNIST model, it was an appropriate task to validate our data flow architecture. mnist/train_mnist_fc.py and mnist/mnist.py represent our training and image processing.

To control the accelerator and exchange data during inference, we implemented a communication layer using a UART-to-AXI bridge[4]. The UART interface enables a Python GUI application running on a PC to communicate with the FPGA over a standard USB. Inside the FPGA, the UART controller translates the incoming

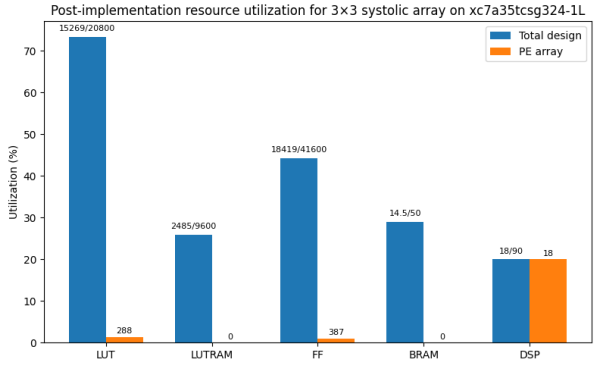

Figure 6: Post-Implementation Resource Utilization for 3x3 Systolic Array on xc7a35tcsg324-1L

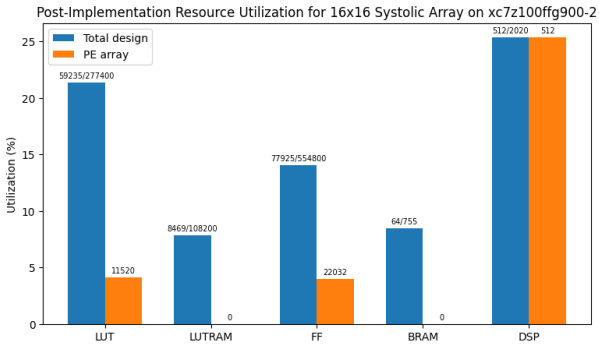

Figure 7: Post-Implementation Resource Utilization for 16x16 Systolic Array on xc7z100ffg900-2

commands into AXI-Lite transactions, enabling direct access to the accelerator from the host. The functions for access and control are described in eyeriss_fpga/mnist/ws_fpga_funcs.py, and an example of function use is shown in /mnist/mnist_gui.py and ws_fpga/scripts/ws_simple_matrix_multiply_test.py. The GUI application described is mnist/mnist_gui.py. The full hardware and software implementation can be found on our GitHub repository at https://github.com/cbarbr/ws_fpga.

4 Implementation Results

After place-and-route, the design achieves a stable 100 MHz global clock frequency without requiring aggressive timing constraints. In this section, we present the timing, resource utilization, and end-to-end performance of the implemented design, followed by an analysis of system bottlenecks and optimization opportunities.

4.1 Resource usage

The final post-implementation utilization shows that the accelerator occupies only a small portion of the available FPGA resources. The systolic array and control logic fit comfortably within the LUT, FF, DSP, and BRAM budgets, aligning with our design goal of minimizing hardware footprint for resource-constrained devices. From Figure 6 and 7, we observe that the majority of LUT resources are consumed by top-level support logic, including the UART interface, debug circuitry, and other infrastructure modules. In contrast, all DSP blocks are utilized exclusively by the PE array, and the total DSP usage is approximately twice the number of PEs, as each PE contains two DSPs.

4.2 Performance

To evaluate real execution latency, we measured the full hardware–software pipeline, including weight writing, activation writing, computation, and result psum reading.

Figure 8 illustrates that the total latency is dominated by UART-AXI-based data transfers. The computation time of the systolic array is nearly constant across matrix sizes, whereas the weight-write, input-write, and psum-read operations grow approximately quadratically with the matrix dimension. For the 16×16 case, more than 98% of the total runtime (2.576 s) results from the UART and

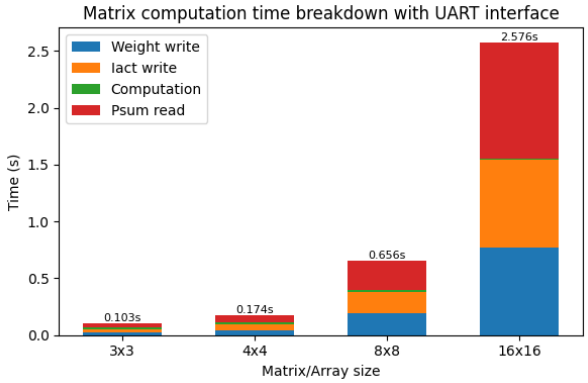

Figure 8: Matrix Computation Time with UART-AXI Interface

AXI communication, with the psum read back becoming the largest component because the psum has 64 bits. And all the computation times for different PE array sizes are the same (0.016 s) because the theoretical computation time (68 cycles * 10 ns = 680 ns, for 16x16 array) is much less than the register reading time through UART to AXI interface.

To eliminate the slow UART interface and the PySerial library, we deployed the 16×16 systolic array design on Zynq Processing System (PS) running Kuiper Linux, which can directly access the PL BRAMs through AXI. Instead of UART, the Python program uses the mmap interface to directly read and write AXI-mapped BRAM addresses.

With the AXI-based host interface, the hardware multiply time for a 16×16 matrix is nearly 25× faster than the UART-AXI-based implementation. However, as illustrated in Figure 9, the overall execution time continues to be dominated by weight and activation write latency as well as psum readback latency, all of which scale with the matrix dimension. In contrast, the compute phase itself accounts for less than 0.2% of the total runtime, indicating that the accelerator's end-to-end performance is fundamentally constrained by communication bandwidth rather than compute throughput.

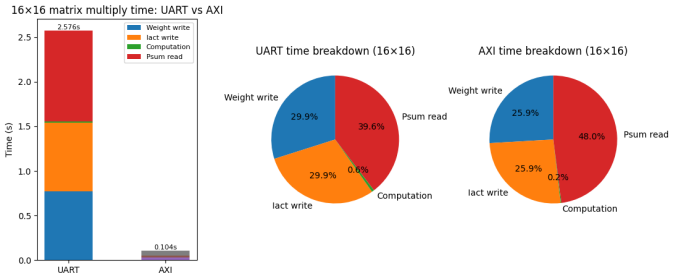

Figure 9: Matrix Computation Time with UART-AXI and AXI Interface for 16x16 Systolic Array

To further hide the memory read and write latency, the system can adopt a higher-frequency AXI clock, use DMA engines for burst transfers, or trigger completion using PS IRQ_F2P interrupts to avoid repeatedly polling the "done" register over AXI.

5 Related Work

5.1 Energy Efficient GeMM-based Convolution Accelerator with On-the-Fly im2col

Fornt et al. [3] demonstrate how GeMM systolic array architecture can be effectively used to perform convolutions. This is done primarily through the inclusion of an im2col system to perform convolution lowering on-chip and avoid the inflated memory cost of im2col as a pre-processing step. This im2col processed data is then fed into a output-stationary systolic array for the GeMM operation. This architecture was able to achieve 1.1 TFLOPS/W when implemented in a 22nm process fitting in an area of 1.1mm^2

This work is similar to ours in it's use of GeMM accelerator hardware capable of performing convolution operations. Our design requires convolution lowering as pre-processing and adding on-chip im2col is a logical and highly beneficial improvement we could make. Our design used a weight stationary array rather than output stationary, which we believe could allow it to achieve better data reuse after convolution lowering due to weights generally being smaller than output and therefor easier to fit in the array.

5.2 Eyeriss v2

Eyeriss v2 is a CNN accelerator optimized to compute on sparse data in a compressed domain [2]. This work uses compressed domain operations to gain significant speedup and energy efficiency gains on sparse networks. Eyeriss v2 uses a row-stationary architecture and hierarchical mesh network to obtain high data reuse and PE utilization for convolution, depth-wise convolution, and fully connected layers. These features are pair well with models that are designed to have a reduced size by being sparse or compact. Optimizing for these types of models allows Eyeriss v2 to achieve high energy efficiency at low latency, especially for sparse models.

This is a fairly different approach from our design, although with a similar goal. Eyeriss v2 takes advantage of extra steps in the training of a model to reduce memory overhead and optimizes for these smaller models. This creates some restrictions on what models Eyeriss v2 can run most effectively, with sparsity especially having a large impact on energy efficiency. Our design relies more on the improved efficiency of using a systolic architecture to get improved overall efficiency. This means that the peak efficiency of our design is much less than that of Eyeriss v2 but does not rely on extra steps in the training of a model to achieve.

Conclusion

In this paper, we describe the FPGA implementation of a weight-stationary systolic array-based GeMM accelerator. Our accelerator was successfully run on an FPGA and used to perform inference with a simple model trained on the MNIST dataset.

Our experiments demonstrate that our design is functionally correct and scalable. Performance characterization revealed that while the systolic core is capable of high throughput, the end-to-end system latency is currently dominated by our host-to-FPGA communication interface.

These results come from the relatively small size of the PE array, which limits the amount of data reuse that can be achieved. In addition, our design was unable to compute GeMM operations on matrices larger than the PE array, which greatly increased the communication overhead for large matrices.

Looking forward, this work establishes a solid foundation for further optimization. Further extensions include an on-chip im2col unit to eliminate off-chip pre-processing. The greatest improvement would be to enable the accelerator to perform large matrix multiplications on a relatively smaller PE array, which would allow for much better utilization of the FPGA's memory and greatly reduce communication latency for larger matrices. Once these bottlenecks are addressed, the proposed architecture offers a promising template for high-performance, low-power NN inference at the edge.

Member Contributions

This project was executed through a highly collaborative workflow. Significant portions of the RTL were co-authored during joint working sessions.

- Maatla Sefawe: RTL Design and Implementation, Verification and Testing, Poster Design and Writeup
- Andrew Keil: RTL Design and Implementation, Verification and Testing in Simulation, Architecture Design of PE and Controller
- Cameron Barbour: Host interface block design, top level integration, synthesis and implementation, python UART driver class, assisted with design of PE and controller, testing and verification over JTAG & UART
- Tong Sing Wu: RTL Design and Implementation, Verification and Testing, Python GUI object, MNIST Training & Image Processing script
- Guanghao Chen: RTL Design and Implementation, Verification and Testing on ZYNQ FPGA for 16x16 PE array

References

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

[2] Yu-Hsin Chen, Tien-Ju Yang, Joel Emer, and Vivienne Sze. 2019. Eyeriss v2: A Flexible Accelerator for Emerging Deep Neural Networks on Mobile Devices. doi:10.48550/arXiv.1807.07928 arXiv:1807.07928 [cs].

[3] Jordi Fornt, Pau Fontova-Musté, Martí Caro, Jaume Abella, Francesc Moll, Josep Altet, and Christoph Studer. 2023. An Energy-Efficient GeMM-Based Convolution Accelerator With

On-the-Fly im2col. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 31, 11 (Nov. 2023), 1874–1878. doi:10.1109/TVLSI.2023.3286122

[4] GHz-WS. 2025. ghz-ws/uart2axi. https://github.com/ghz-ws/uart2axi original-date: 2025-09-14T08:11:02Z.

[5] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2015. Deep Residual Learning for Image Recognition. doi:10.48550/arXiv.1512.03385 arXiv:1512.03385 [cs].

[6] Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre-luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann, C. Richard Ho, Doug Hogberg, John Hu, Robert Hundt, Dan Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Daniel Killebrew, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon, James Law, Diemthu Le, Chris Leary, Zhuyuan Liu, Kyle Lucke, Alan Lundin, Gordon MacKean, Adriana Maggiore, Maire Mahony, Kieran Miller, Rahul Nagarajan, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, Andy Phelps, Jonathan Ross, Matt Ross, Amir Salek, Emad Samadiani, Chris Severn, Gregory Sizikov, Matthew Snelham, Jed Souter, Dan Steinberg, Andy Swing, Mercedes Tan, Gregory Thorson, Bo Tian, Horia Toma, Erick Tuttle, Vijay Vasudevan, Richard Walter, Walter Wang, Eric Wilcox, and Doe Hyun Yoon. 2017. In-Datacenter Performance Analysis of a Tensor Processing Unit. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*. ACM, Toronto ON Canada, 1–12. doi:10.1145/3079856.3080246

[7] Mahdi Kalbasi. 2024. Architectural Insights: Comparing Weight Stationary and Output Stationary Systolic Arrays for Efficient Computation. In *2024 15th International Conference on Information and Knowledge Technology (IKT)*. 146–150. doi:10.1109/IKT65497.2024.10892683 ISSN: 2476-2180.

[8] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. 2012. ImageNet Classification with Deep Convolutional Neural Networks. In *Advances in Neural Information Processing Systems*, Vol. 25. Curran Associates, Inc. https://proceedings.neurips.cc/paper_files/paper/2012/hash/c399862d3b9d6b76c8436e924a68c45b-Abstract.html

[9] Kung. 1982. Why systolic architectures? *Computer* 15, 1 (Jan. 1982), 37–46. doi:10.1109/MC.1982.1653825

[10] Vivienne Sze, Yu-Hsin Chen, Tien-Ju Yang, and Joel S. Emer. 2017. Efficient Processing of Deep Neural Networks: A Tutorial and Survey. *Proc. IEEE* 105, 12 (Dec. 2017), 2295–2329. doi:10.1109/JPROC.2017.2761740

[11] Chen Zhang, Peng Li, Guangyu Sun, Yijin Guan, Bingjun Xiao, and Jason Cong. 2015. Optimizing FPGA-based Accelerator Design for Deep Convolutional Neural Networks. In *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. ACM, Monterey California USA, 161–170. doi:10.1145/2684746.2689060