MXITA: Design and Implementation of Microscaling Integer Accelerator for Neural Networks

An exploration of multidimensional systolic arrays

Li Ou Hu

TU Delft

MXITA: Design and Implementation of Microscaling Integer Accelerator for Neural Networks

An exploration of multidimensional systolic arrays

Master's Thesis

by

Li Ou Hu

to obtain the degree of Master of Science
at the Delft University of Technology
to be defended publicly on September 23, 2025 at 09:00

Thesis committee:

Supervisor:	Prof. dr. ir. Georgi Gaydadjiev
Daily supervisors:	Gamze Islamoglu
	Philip Wiese
Project Duration:	February, 2025 - September, 2025
Student number:	5236541

An electronic version of this thesis is available at `http://repository.tudelft.nl/`.

Preface

I would like to wholeheartedly thank Prof. Dr. Luca Benini and Prof. Dr. Georgi Gaydadjiev for giving me the chance to pursue my Master's thesis project at ETH Zurich. This experience has truly been life-changing, and I will always remain deeply grateful to them for their support and trust.

I am also grateful to my supervisors Gamze Islamoglu, Philip Wiese and (in extension) Philippe Sauter for their guidance, discussions, and assistance during the course of this project. Special thanks to Doruk Bekatli for his help on the back-end implementation, and the Microelectronics Design Center for their suggestions and improvements.

Finally, I would like to thank my family and friends for their unwavering support in me during this journey. This achievement would not have been possible without them.

Contents

Nomenclature

List of Abbreviations

ALU	Arithmetic Logic Unit
AXI	Advanced eXtensible Interface
DMA	Direct Memory Access
DNN	Deep Neural Network
DUT	Design Under Test
FIFO	First-In First-Out
FSM	Finite State Machine
GPU	Graphics Processing Unit
HBM	High Bandwidth Memory
HWPE	Hardware Processing Engine
I/O	Input/Output
LLM	Large Language Model
MAC	Multiply-Accumulate
MHA	Multi-Head Attention
ML	Machine Learning
MMM	Matrix-Matrix Multiplication
MVM	Matrix-Vector Multiplication
MX	Microscaling
NPU	Neural Processing Unit
PE	Processing Element
PPA	Performance Power Area
SIMD	Single-Instruction Multiple-Data
SOTA	State Of The Art
SRAM	Static Random Access Memory
TCDM	Tightly Coupled Data Memory
TOPS	Tera Operations Per Second
TPU	Tensor Processing Unit
VLSI	Very Large-Scale Integration

List of Figures

List of Tables

1

Introduction

The rapid advancement of machine learning algorithms has created a growing demand for specialized high-performance computing. For example, natural language models have shown to exhibit power-law scaling in loss with respect to the model size [1], highlighting the increasing computational requirements. While Moore's Law [2] historically predicted exponential growth in transistor density, this trend is approaching physical and practical limitations. As a result, meeting future computational demands will require innovations not only at the software application level but also in the design and architecture of computing hardware. Specifically, at the software application level, the neural network architecture may be changed and quantized in order to reduce the computational demands. Furthermore, hardware architectures can be developed such that patterns found within the computational flow of neural networks can be accelerated as efficiently as possible. This includes for example reuse of data to reduce memory accesses or exploiting parallelism for more efficient computations.

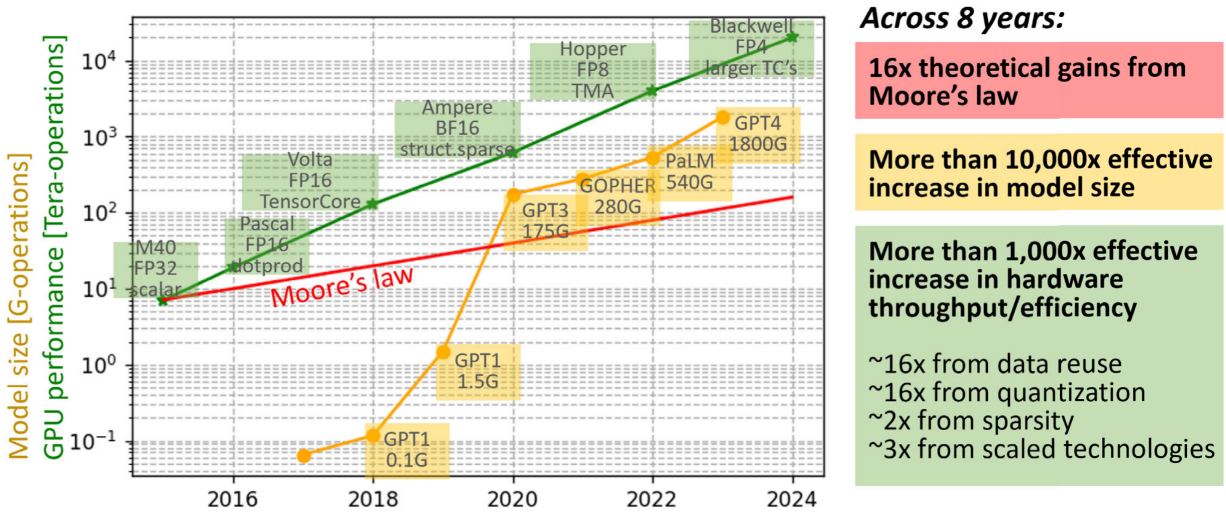


Figure 1.1: Evolution of machine learning model size, GPU performance and Moore's law across the past decade. Figure obtained from [3]

To reduce the computational and storage costs of deep learning models, low bit width formats such as `INT8` and `FP16` are widely adopted in AI-focused hardware, including Graphics Processing Units (GPUs), Tensor Processing Units (TPUs) [4], and edge inference devices. The main method to reduce neural network model sizes is through quantization, which involves transforming model weights represented by a high-precision floating-point format (i.e. FP32) to a lower precision such as 8-bit integer or floating-point format. This can be done by for example quantizing the model after training (post-training quantization), or with additional fine-tuning of the model after quantization (quantization-aware training). Such quantized neural networks rely on scaling factors at the neural network layer level to manage the limited dynamic range, but often fall short in preserving accuracy.

A more effective quantization strategy involves sharing scaling factors across fine-grained sub-blocks

within the data format. Microscaling (MX) quantization [5] introduces a family of micro-scaled data types that reduce memory consumption while preserving accuracy levels comparable to FP32 for both training and inference. However, these formats require specialized mixed integer–floating point arithmetic to enable efficient computation.

In neural network inference, matrix multiplications typically dominate execution time. To accelerate these operations, two major hardware paradigms have been explored: SIMD architectures and systolic arrays. Systolic arrays are often favored because of their simple and regular structure, built from small, fast processing elements, and their ability to provide an efficient balance between computation and I/O bandwidth [6]. Despite these advantages, only limited research has addressed the efficient mapping of MX quantized matrix multiplications onto systolic arrays.

This thesis investigates the design and evaluation of systolic-array architectures tailored for MX quantized matrix multiplications. The focus is placed on both the MX data format and corresponding hardware design techniques required to accelerate MX arithmetic efficiently, with a specific emphasis on neural network workloads.

2

Background and Related Work

2.1. Microscaling (MX) data format

In the MX data format [5], a data format is proposed where a per-block scaling factor S is shared with k narrow integer or float-point individual elements. The MX specification supports blocks with 4-bit to 8-bit wide FP elements, or INT8 elements, where elements a in a block share an 8-bit exponent scale value S as illustrated in Fig. 2.1. With this format, the memory consumption can be reduced compared to only using a FP32 datatype while maintaining a competitive neural network inference and training accuracy. In this thesis, we focus on the MXINT8 data format from the MX specification, as MXINT8 has shown to be a compelling alternative to FP32 for direct-cast inference [5].

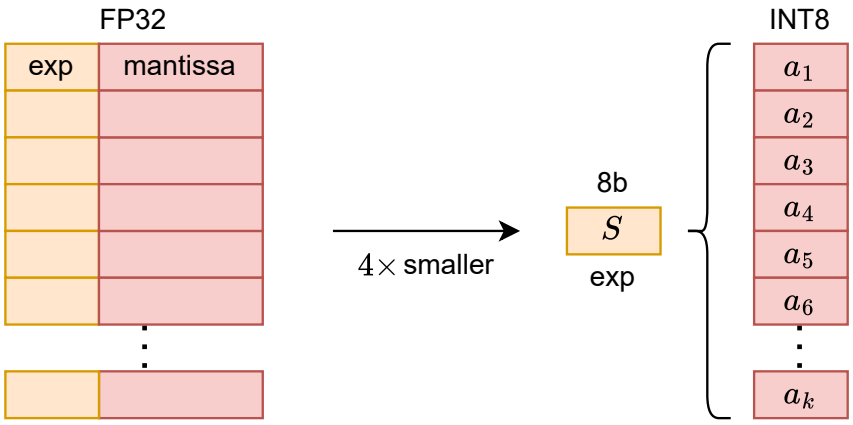


Figure 2.1: Comparison of FP32 and MXINT8 data format, where a 8-bit block scale S is shared across k INT8 elements. Depicted in orange is the exponent/scale and in red the mantissa/integer element.

The memory reduction gains from using an MXINT8 data format compared FP32 can be quantified based on the MX block size k. For instance, the memory consumption for a single MXINT8 block of k elements will require $8k + 8$ bits, while the same amount of elements in FP32 will require 32 bits. Therefore, the memory reduction gains can be defined as $\frac{32k}{8k+8}$, which is plotted in Fig. 2.2. For larger block sizes $k \to \infty$, it can easily be seen that the theoretical memory reduction gain is $4\times$, as an INT8 element is 4 times smaller than a FP32 element. However, it is imperative to determine an optimal block size k to balance the memory reduction tradeoff with the neural network inference accuracy. For this, a range of block sizes $8 \leq k \leq 32$ can therefore chosen for a memory reduction $3.56\times$ to $3.88\times$. Nevertheless, prior work suggests that $k = 16$ offers the best trade-off between accuracy retention and memory efficiency [7].

While MX quantization provides a significant amount of memory reduction for minimal neural network inference accuracy loss, arithmetic with MX datatypes involve computations that require specialized hardware in order to be executed efficiently. For example, a dot product between two MX blocks $2^{S_a}\vec{a} \cdot 2^{S_b}\vec{b}$ can be represented as $2^{S_a+S_b}c = 2^{S_a+S_b}(\vec{a} \cdot \vec{b})$. While $\vec{a} \cdot \vec{b}$ can be computed in their native data format, for vectors longer than k where the vector will consist of multiple blocks i.e. $\vec{a} = \{2^{S_{a,1}}\vec{a_1}, 2^{S_{a,2}}\vec{a_2}\}$ will require inter-block arithmetic where the dot-product results are first casted to FP32. For example, the dot product

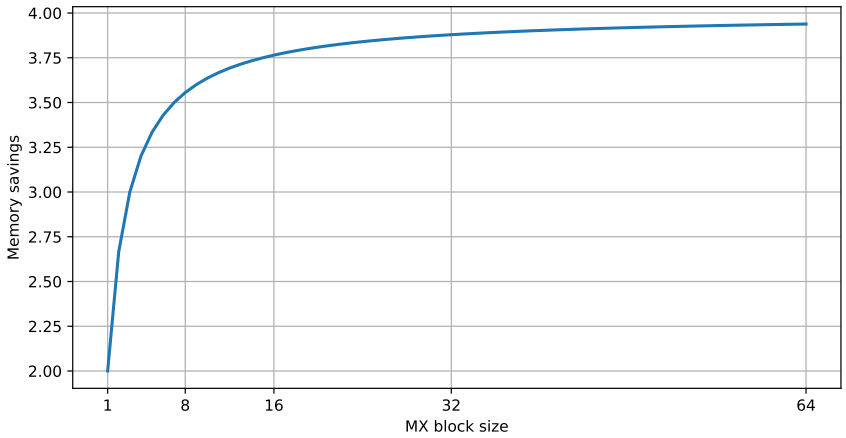


Figure 2.2: MXINT8 memory savings w.r.t. FP32 for MX block sizes $64 \geq k \geq 1$

in this case will be represented as $c_{\text{FP32}} = 2^{S_{a,1}+S_{b,1}}\vec{a_1}\vec{b_1} + 2^{S_{a,2}+S_{b,2}}\vec{a_2}\vec{b_2}$, where the addition of the partial dot product results has to be performed after conversion to a common data format such as FP32. For quantizing a block of FP32 results, the scaling value S_c is set to be the largest power-of-two smaller than max$(\vec{c})$.

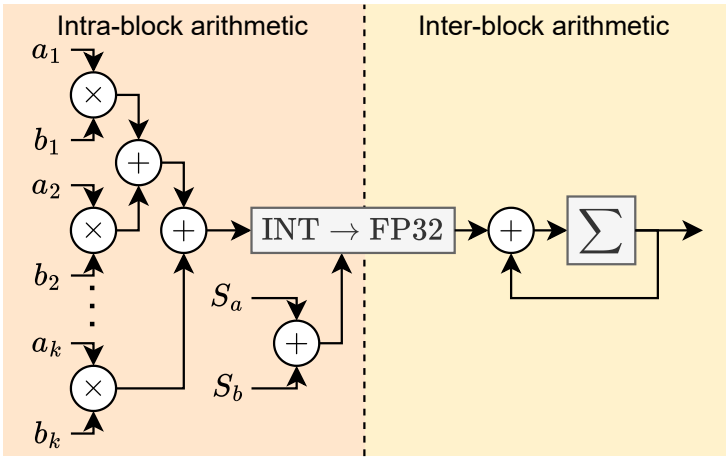


Figure 2.3: Dot product between two vectors in MXINT8 data format $\vec{a} \cdot \vec{b}$. After computing the dot product of a block in its element-wise INT8 data format, the result is casted to FP32 and then scaled with S_a and S_b. This scaled FP32 result is then accumulated in order to compute the dot product across multiple MX blocks.

In this thesis, we focus on MX-quantized neural network inference, where matrix multiplications typically represent the dominant computational kernel. An example of an MXINT8 matrix multiplication is shown in Fig. 2.4, involving matrices A and B, which in this work will be referred to as the *input* and *weight* matrices, respectively.

2.2. Systolic arrays

Matrix multiplications are commonly accelerated using systolic arrays. Systolic arrays, a concept introduced by Kung [6], are computing architectures composed of simple Processing Elements (PEs) interconnected in a manner that allows data to flow through the system in a ”rhythmic” fashion, as illustrated in Fig. 2.5. In a typical 2-dimensional configuration, the PEs are arranged in a grid and connected both vertically and horizontally. The input x (red) and weight w (blue) matrix operands are fed into the array via shift registers, which introduce skewing to align operands appropriately across the PEs. The resulting output matrix (green), denoted as c, is time-multiplexed and skewed across the rows. Skew buffers at the output

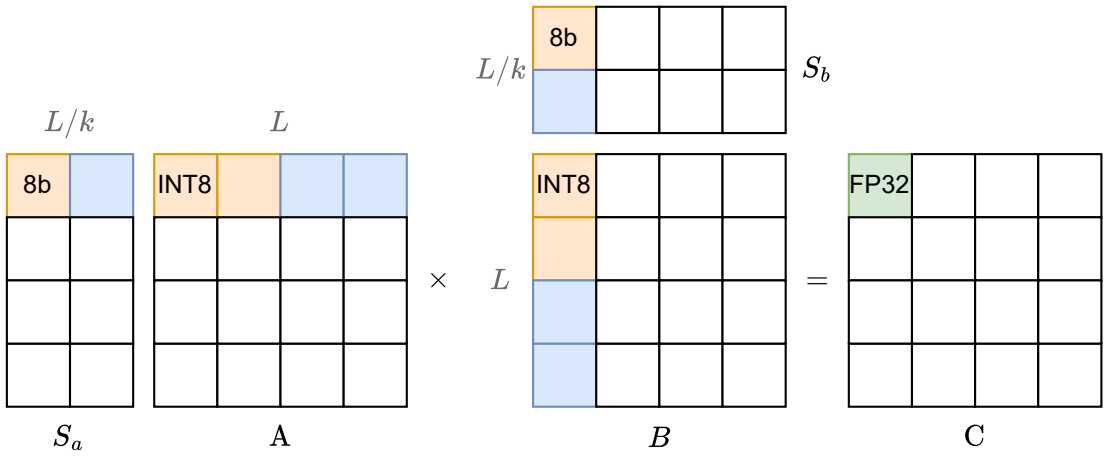


Figure 2.4: Matrix multiplication of MX quantized matrices A and B with respective scales S_a, S_b for an MX block size $k = 2$ and inner dimension $L = 4$

stage serve to remove the temporal skew introduced during data propagation, such that the final results are correctly aligned.

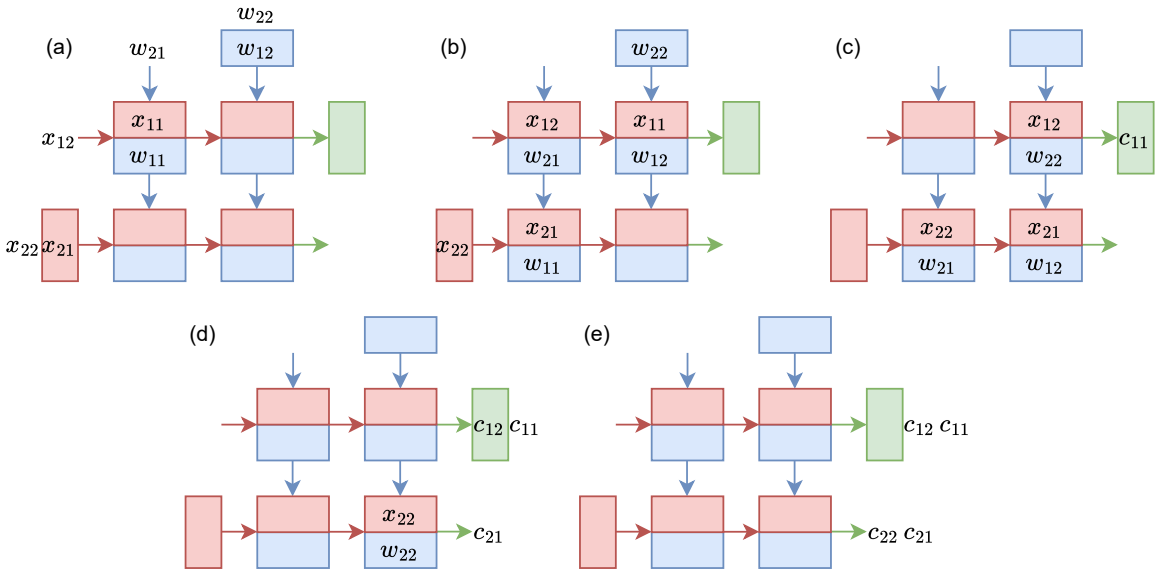


Figure 2.5: Dataflow of the input x, weight w and output c matrix within a 2×2 output stationary systolic array

This design offers several advantages that make it particularly well-suited for machine learning workloads. First, high throughput and parallelism are achieved as multiple PEs can perform computations simultaneously. Second, the propagation of operands through the systolic array enables reuse of values, thereby reducing the number of accelerator memory accesses required. Finally, the regular and modular structure of systolic arrays allows for scalability of the systolic array size, making them highly amenable to Very-Large Scale Integration (VLSI) implementations.

Output Stationary Processing Element (PE)
In this work, we focus on output-stationary systolic arrays, where each PE performs Multiply-Accumulate (MAC) operations over a single output matrix element while the input x and weight w operands propagate through the PE, as illustrated in Fig. 2.6. As the input x and weight w flow sequentially through the processing element, an INT8 MAC operation $o_{\text{next}} = xw + o$ is performed using the current accumulated value o.

Once the PE has completed the accumulation for an entire dot product, the resulting output o is transmitted to the output matrix. This transfer is initiated by a ”pop” signal, which propagates throughout the systolic array, as illustrated in Fig. 2.7. The pop signal serves a two functions: it acts as a select signal for multiplexing the output o, and simultaneously indicates the validity of the output through a ”write enable” signal.

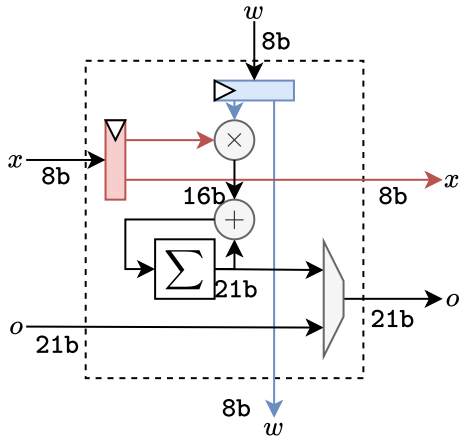


Figure 2.6: Output stationary processing element (PE) for INT8 MAC operations

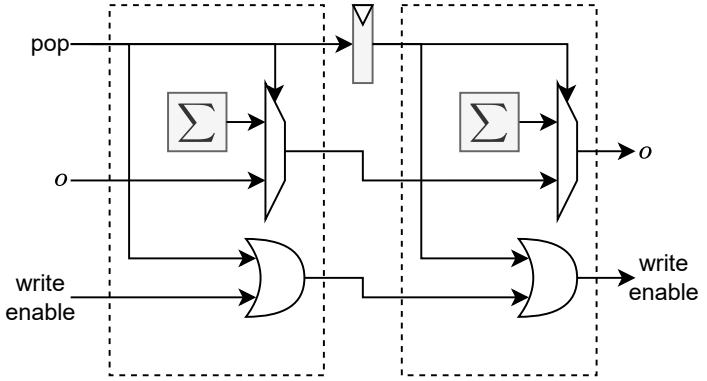


Figure 2.7: Control flow and output multiplexing for a row of PEs

As the number of PEs in a row increases, the number of multiplexers in a chain increases. This chain of multiplexer will become the critical path of the entire design if the systolic array scales up. As the pop signal over a row of PEs is a one-hot vector, the multiplexer chain can instead be optimized to AND/OR gates as illustrated in Fig. 2.8.

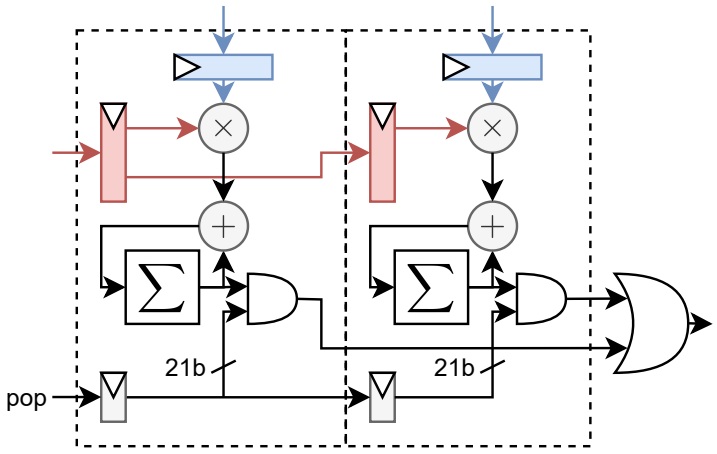


Figure 2.8: Optimized output selection design. The pop signal is extended to match the integer output bit width, and passed as an operand to the AND-gate together with the PE sum.

Multidimensional systolic arrays

Output-stationary PEs can be interconnected in different systolic array arrangements. Such systolic array architectures for matrix multiplication with generic datatypes such as FP32 or INT8 will be presented. These systolic array designs can be classified according to the degree of operand reuse and the broadcasting employed, where we introduce the concept of ”multidimensional” systolic arrays. The most basic form is the 0-dimensional systolic array, illustrated in Fig. 2.9, which exhibits neither operand reuse nor operand broadcasting. In this design, each PE is responsible for accumulating only a single scalar value, corresponding to one element of the output matrix (green), i.e., a 0-dimensional ”dot”. During each computation

cycle, the PE receives and processes elements from the inner dimension L of the input (red) and weight (blue) matrices to perform the partial accumulation.

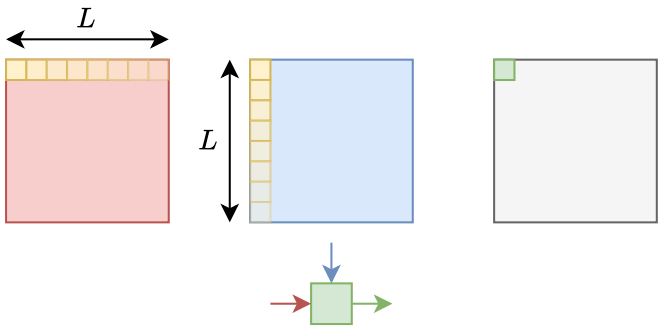


Figure 2.9: 0-Dimensional systolic array

Extending this concept, a 1-dimensional systolic array is constructed by arranging multiple PEs into a row of length M, as shown in Fig. 2.10. This organization enables reuse of the input matrix across M processing stages. To facilitate this reuse, shift registers are introduced for the weight matrix, ensuring that its elements are supplied in a skewed manner relative to the shared input matrix. The outputs produced by the PEs are then multiplexed to form the final result, which requires M cycles to fully propagate through the array.

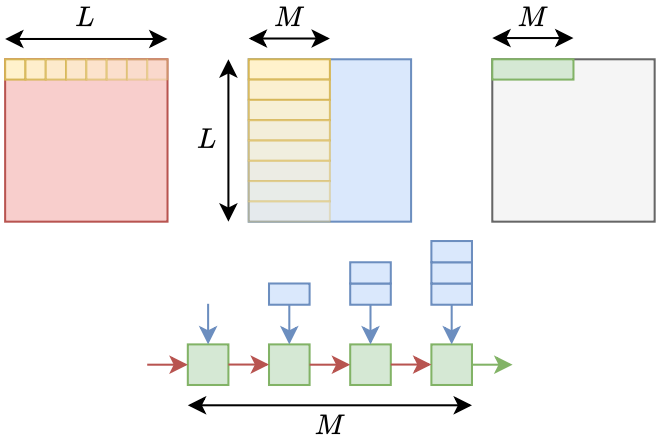


Figure 2.10: 1-Dimensional systolic array with $M = 4$

The most conventional and widely adopted architecture for matrix multiplications is the 2-dimensional systolic array, illustrated in Fig. 2.11. In this design, PEs are organized into an $M \times N$ grid, allowing for simultaneous computation of multiple output elements. The structure exploits data reuse by propagating the input matrix elements along M columns and the weight matrix elements along the N row. As a result, each PE receives a unique pair of operands that are reused across the array, thereby improving computational efficiency and reducing memory bandwidth requirements.

To support this data movement, skew buffers are employed, similar to the 1-dimensional case. In a 2-dimensional systolic array, skewing is required for both the input and weight matrices to ensure proper temporal alignment of operands as they traverse the systolic array. Additionally, shift registers are necessary for the output matrix to correctly synchronize partial sums generated across multiple PE rows.

An extension of the 2-dimensional systolic array is the 3-dimensional systolic array, illustrated in Fig. 2.12. This architecture is constructed by broadcasting the weight matrix operand across P parallel 2-dimensional systolic arrays. Alternatively, the same construction can be achieved by broadcasting the input matrix operand instead. Each of these P 2-D systolic arrays is responsible for producing $M \times N$ output matrix elements, which each form a 2-dimensional ”plane” of the output space. By stacking P planes,

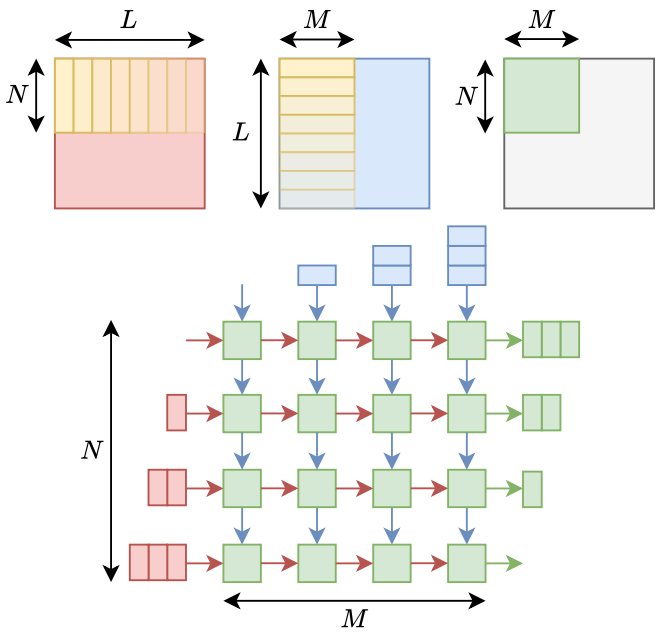


Figure 2.11: 2-Dimensional systolic array with $(M, N) = (4, 4)$

the overall structure produces a 3-dimensional output "volume", with the additional P axis serving as the third dimension. Notably, the 3-dimensional organization has less sequential logic for skewing its operands, compared to its 2-dimensional counterpart. In particular, when the weight matrix is broadcast across the P planes, as shown in Fig. 2.12, the broadcast effectively acts as a skip connection, directly delivering operands to subsequent PEs. This bypass mechanism eliminates the need for weights to propagate further along the N dimension, thereby reducing the buffering requirements compared to lower-dimensional systolic array designs.

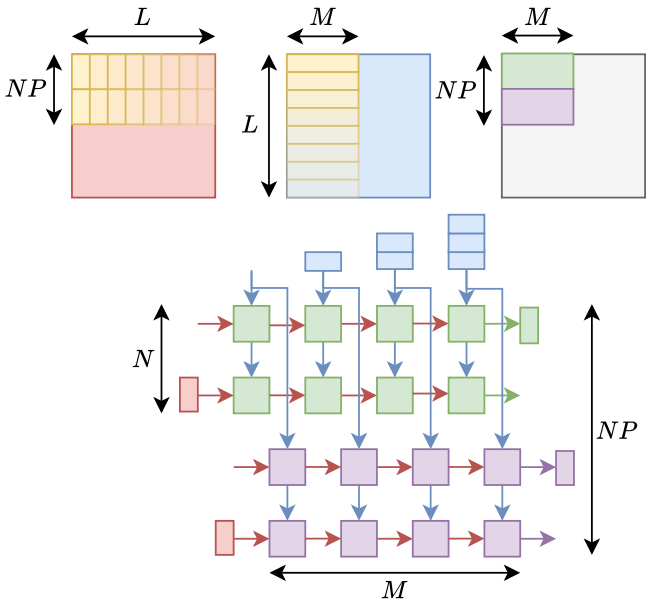


Figure 2.12: 3-Dimensional systolic array with $(M, N, P) = (4, 2, 2)$

Extending the concept further, a 4-dimensional systolic array can be realized by broadcasting not only the weights, as in the 3-dimensional case, but also the input matrix operand. In this design, both operands are distributed across a 2-dimensional grid of 2-dimensional systolic arrays, thereby further

reducing the amount of sequential logic associated with operand propagation. Moreover, the architecture enables propagation of all data through the array in only M cycles, attaining a peak throughput of NPQ output elements per cycle. While the 4-dimensional systolic array offers significant advantages in terms of parallelism and throughput scaling, it also introduces new implementation challenges. In particular, the routability of the design becomes a primary concern during the place-and-route stage of the physical design. Because both inputs and weights are broadcast simultaneously, their data paths may overlap, which scales with the additional broadcast dimensions P and Q. This increased wiring complexity may lead to congestion in its physical implementation, which can offset the theoretical area reduction.

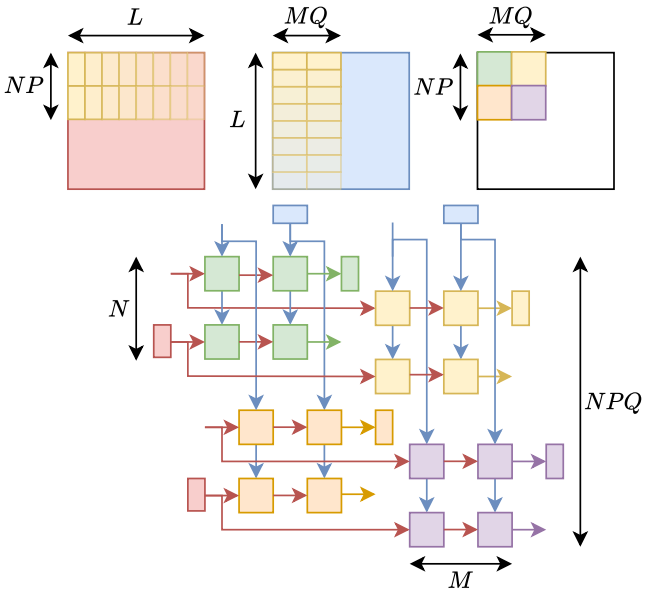


Figure 2.13: 4-Dimensional systolic array with $(M, N, P, Q) = (2, 2, 2, 2)$

A 5-dimensional systolic array (Fig. 2.14) can be constructed by adding a stationarity R in the inner dimension L. In the hardware implementation, it represents turning each PE into a dot-product and accumulate unit. Each PE receives R input and weight elements per cycle, and requires R times less cycles to produce the final output matrix result.

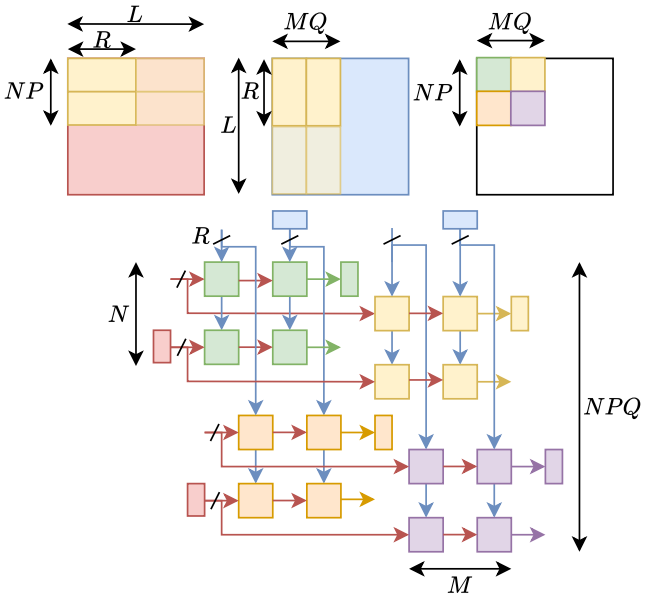


Figure 2.14: 5-Dimensional systolic array with $(M, N, P, Q, R) = (2, 2, 2, 2, 4)$

2.3. Transformer Neural Networks

Transformers, introduced by Vaswani et al. [8], depart from recurrent and convolutional architectures by leveraging self-attention to capture long-range dependencies. After surpassing RNNs in accuracy at similar cost, they have been adopted across text, image, audio, and video, highlighting both their general-purpose nature and the pressing need for efficient hardware acceleration.

In this thesis, we use the transformer as a representative workload, focusing specifically on its core computational kernel: Multi-Head Attention (MHA), illustrated in Fig. 2.15. Given an input of size $S \times E$, where S is the sequence length and E the embedding dimension, three linear transformations generate the Query (Q), Key (K), and Value (V) matrices. Each has size $S \times P$, with P denoting the projection dimension, and these transformations are realized as dense matrix multiplications between the input and learned weight matrices.

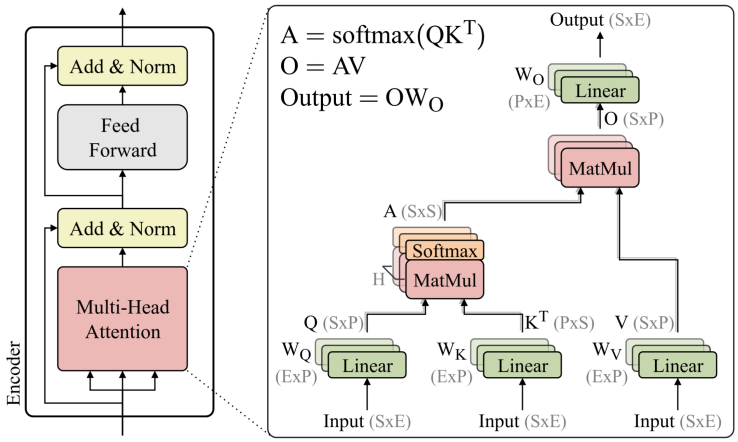


Figure 2.15: Tranformer encoder and multi-head attention block. S: Sequence length, E: Embedding size, P: Projection space, H: Number of heads. Figure obtained from [9]

The first major operation in attention is the computation of the attention scores through the matrix multiplication $Q \times K^\intercal$, producing an $S \times S$ matrix A, which is in turn followed by a softmax function to normalize the scores into probabilities. After computing the softmax of the attention matrix A, it is multiplied by the Value matrix V to generate the weighted output, a second large matrix multiplication. This step distributes the attention scores to the input token representations. Multi-head attention extends this computation across multiple heads, each with independent sets of Q, K, and V projections. The outputs of all heads are then concatenated and passed through a final linear projection, producing an output of size $S \times E$ that matches the input dimensions.

Within the multi-head attention (MHA) layer, illustrated in Fig. 2.15, it should be noted that the softmax operation is performed on the $Q \times K^\intercal$ result, as well as the layer normalization being applied to both the MHA output and the feed-forward network. The softmax function σ, defined in Equation 2.1, is a widely used activation function in machine learning workloads. An analysis of these auxiliary functions is crucial, as these comes with their corresponding computational requirements and data dependencies.

Given an input vector **z**, the softmax is computed by exponentiating each element z_i, followed by normalizing each exponentiated term e^{z_i} with the sum of all exponentiated elements $\sum_{j=1}^{K} e^{z_j}$. It is important to note that the exponential operation constitutes a computationally expensive step, as it requires floating-point arithmetic. Although prior works have investigated numerical approximation techniques to reduce the computational burden of the exponential, such methods are beyond the scope of this thesis.

$$\sigma(\mathbf{z})_i = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}} \tag{2.1}$$

Furthermore, the computation of the normalization term $\sum_{j=1}^{K} e^{z_j}$ introduces a data dependency on the entire input vector **z**. This dependency implies that a complete vector (i.e., either a row or a column) of the output matrix must be computed before the softmax operation can be initiated.

3

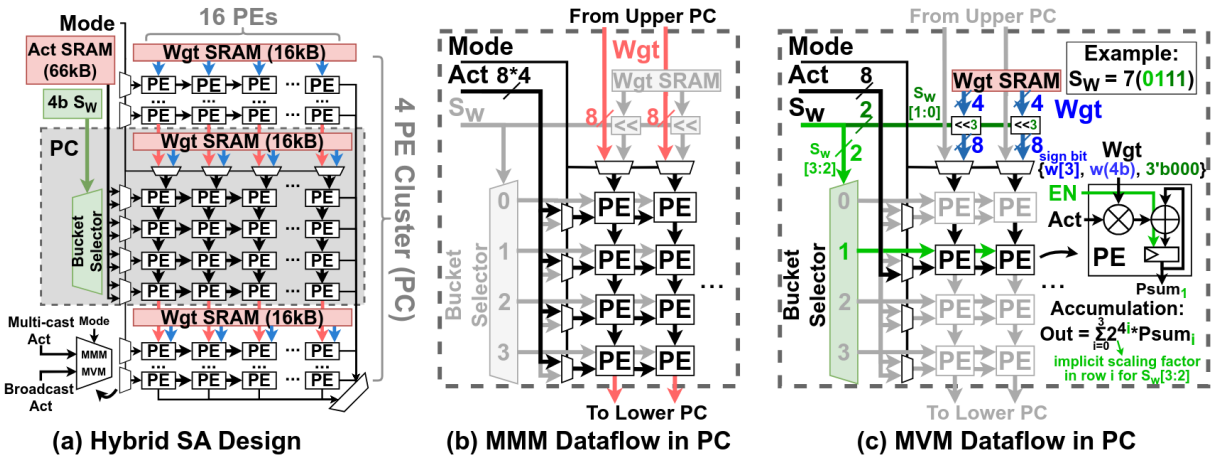
Related work

3.1. Hybrid Systolic Array (HSA)

The Hybrid Systolic Array (HSA) [10], proposed by Chen et al., is a systolic array architecture designed to accelerate both MMMs and MVMs, with a particular focus on workloads associated with LLM. In LLM, the prefill stage predominantly involves MMMs, whereas the decode stage is dominated by MVMs. Given that the decode stage accounts for approximately 80% of the runtime, there is a need for accelerators optimized for MVMs operations.

In HSA, the prefill stage leverages INT8 activation and weight matrices, during which the array is configured as a standard output-stationary systolic array, as illustrated in Fig. 3.1b. For the decode stage, the HSA can be reconfigured to perform MVMs using INT8 activation vectors and MXINT4 weight matrices (Fig. 3.1c). In this mode, the MXINT4 weights are de-quantized to INT8 prior to the MAC operation within each PE. This de-quantization is achieved by shifting the weights and enabling the appropriate “bucket” (or row of PEs) based on the most significant bits (MSBs) of the scale.

To improve PE utilization, the HSA is partitioned into PE clusters (PCs), as depicted in Figure 3.1a. This arrangement allows the activation vector to be broadcast across multiple clusters, increasing throughput with the support of several SRAM blocks, which collectively occupy approximately 30% of the total design area. Finally, Chen et al. also propose more efficient hardware for computing various post-processing functions; however, the design of such computational units fall outside the scope of this thesis.



(a) Hybrid SA Design (b) MMM Dataflow in PC (c) MVM Dataflow in PC

Figure 3.1: Hybrid Systolic Array (HSA) architecture. Figure obtained from [10]

It should be noted that the HSA does not achieve peak memory bandwidth utilization or full PE utilization during the MMM and MVM dataflows, respectively. In the MMM dataflow, the weight SRAMs are effectively underutilized because all PEs are connected to form a single large systolic array. As a result, only one SRAM module is actively accessed, leaving the remaining weight SRAM modules idle and causing suboptimal memory bandwidth utilization.

During the MVM dataflow, total PE utilization is limited to only 25%, as illustrated in Fig. 3.1c, due to the operation of the bucket selector. By selecting only a single row of PEs per cycle, the remaining rows remain idle, preventing the accelerator from achieving its maximum throughput.

Finally, the flexibility of HSA in supporting different MX block sizes is constrained by the row length of the systolic array. When the length of an MX block is shorter than the systolic array row length, the array must be further partitioned ”vertically” into separate column sets, introducing additional complexity and limiting scalability.

3.2. Precision-Scalable Hardware

The precision-scalable hardware architecture proposed by Cuyckens et al. [11] incorporates multiple MX formats within each PE, as shown in Figure 3.2. The overall resource usage associated with supporting these multiple MX formats is reduced by sharing arithmetic units and by reorganizing the presentation of data to the MAC unit for different MX configurations. Each PE performs a multiplication operation in its native element-wise data type, after which its results are gradually aggregated into the L1 and L2 adders in FP32 format. This FP32 result is then accumulated in each PE, requiring an FP32 adder for each MAC unit, which take up at least 27% of the total design area.

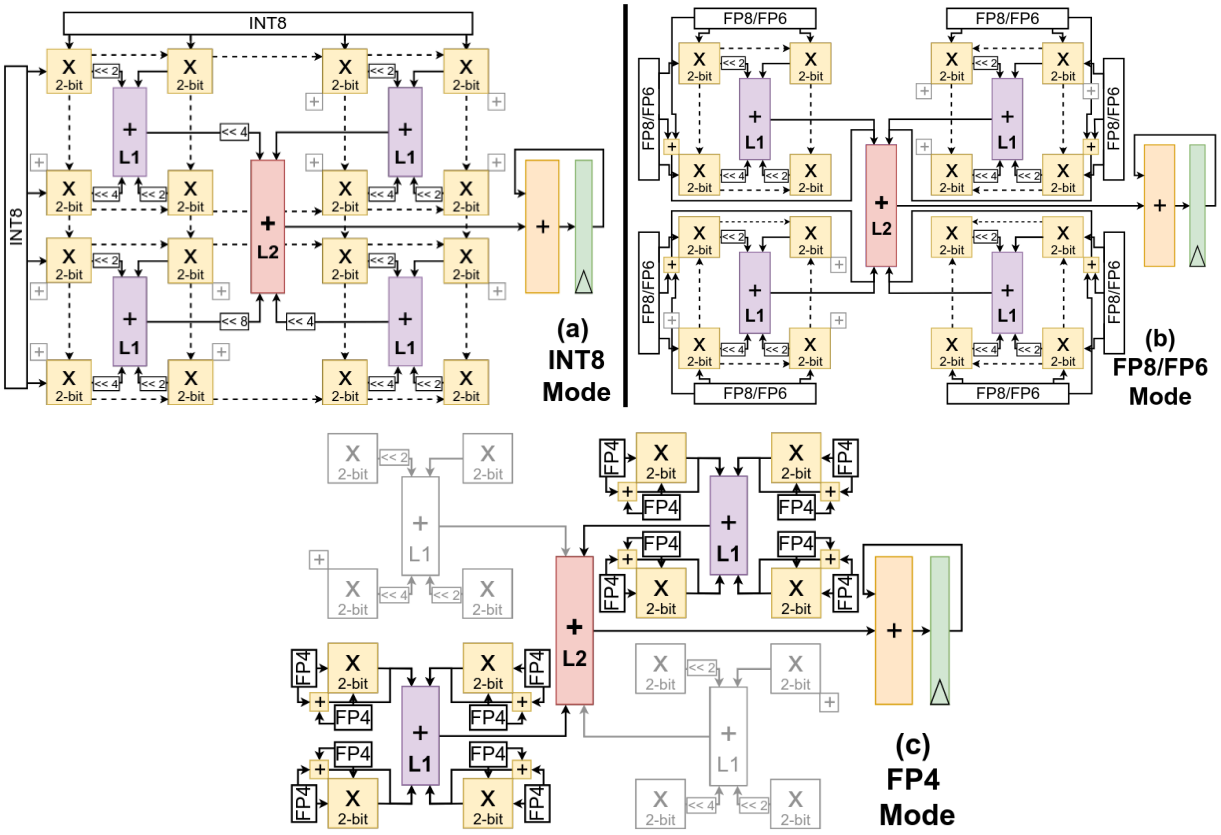


Figure 3.2: Precision-scalable MAC unit for INT8 (a), FP8/FP6 (b) and FP4 (c), respectively. The multiplication units are indicated in yellow, the L1 adder in purple, and the L2 adder in red. Figure obtained from [11]

These PEs are organized into a systolic array, as illustrated in Figure 3.3, specifically designed to accelerate both learning and inference workloads in robotics applications. By implementing square blocks for the tiling of MMM operands, the memory footprint required for performing matrix transposes can be significantly reduced.

Although the precision-scalable MACs provide greater flexibility by supporting multiple MX data formats, the results reported by Cuyckens et al. [11] indicate that the MXINT8 format consistently achieves near-optimal performance in terms of neural network inference and training accuracy compared to other 8-bit MX

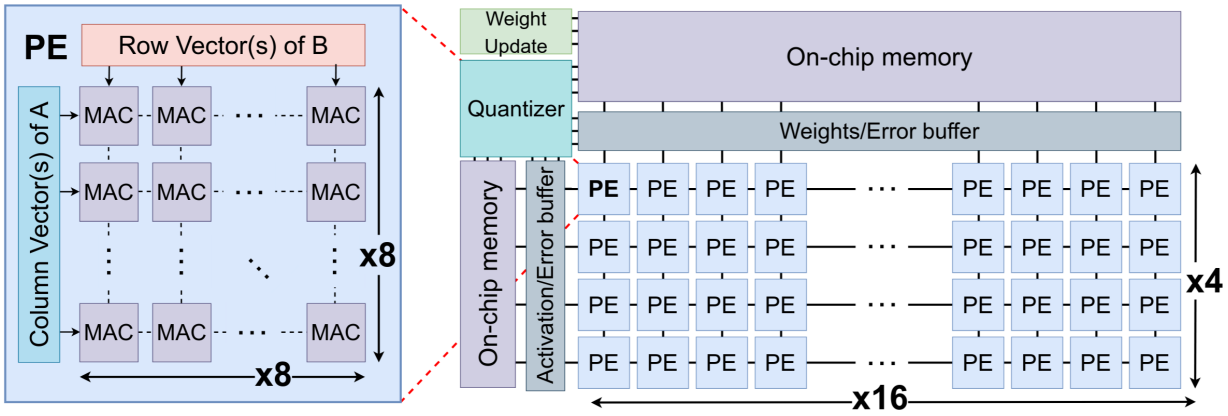


Figure 3.3: PE array, that handles the multiplication of two 64-element square blocks. The design is implemented with 4×16 PE arrays. Figure obtained from [11]

formats (i.e. E4M3 MXFP8 or E5M2 MXFP8). This brings into question the need for supporting different MX formats. Consequently, the additional area overhead required to support multiple MX formats can be justified primarily by the memory savings achieved through smaller MX formats, which must be balanced against potential reductions in training and inference accuracy.

3.3. Jack Unit

Similar to the precision-scalable hardware proposed by Cuyckens et al. [11], the Jack Unit [12] features a MAC unit that supports integer, floating-point, and MX data formats through hardware reuse across formats. As illustrated in Fig. 3.4, accumulation of the output matrix is performed in bfloat16 rather than FP32. While this approach lowers area consumption, it introduces a trade-off in reduced numerical precision.

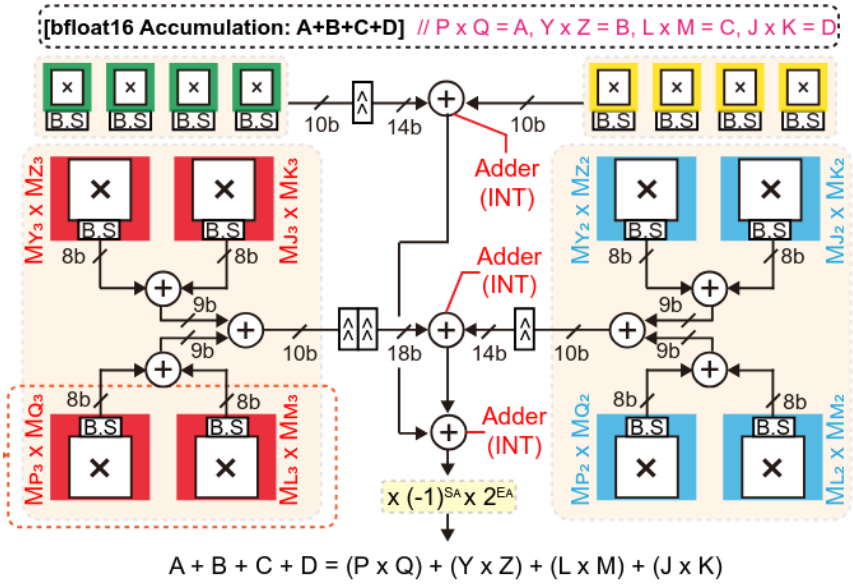


Figure 3.4: Computational flow of Jack Unit for bfloat16 accumulation of mantissas. Figure obtained from [12]

3.4. Tensor Processing Unit (TPU)

The TPU [13], or more generally a NPU, is designed to accelerate matrix multiplications and convolutions, which constitute the most computationally intensive operations in DNN. As illustrated in Figure 3.5, the NPU comprises a systolic array, a vector processing unit, a set of vector registers, and SRAM buffers that store both instructions and vector data.

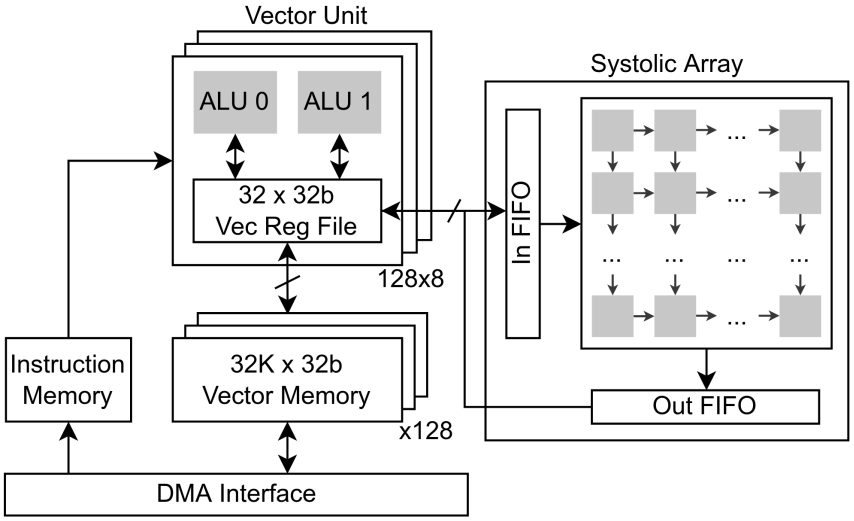


Figure 3.5: Block diagram of a NPU core. Figure obtained from [14]

The SRAM buffers are populated through DMA operations, which execute independently of the ALU core pipeline. This approach allows computation to be overlapped with data movement between on-chip SRAM and off-chip HBM. Within the vector units, multiple SIMD lanes access the vector memory via load/store instructions and perform computations using the vector register files. The vector units also manage data transfers from the vector register file to the systolic array FIFO buffers, ensuring efficient feeding of the processing elements.

A key advantage of the NPU lies in its ability to perform general post-processing computations within the vector units, unlike specialized hardware that lacks programmability. Furthermore, reuse of data elements within the vector memory and vector register files can significantly reduce off-chip data transfers. However, current NPUs do not support MX matrix multiplications.

4 Architecture

In this chapter, we present the hardware architecture of MXITA, with its high-level modules illustrated in Fig. 4.1. The MXITA accelerator is then integrated with a RISC-V processor cluster.

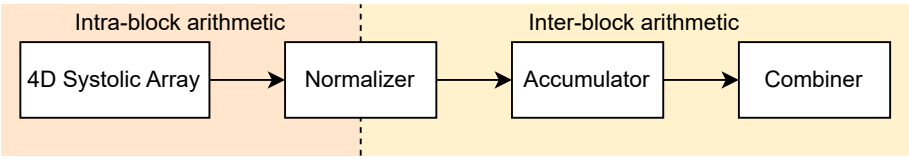


Figure 4.1: High-level overview of the MXITA architecture. Intra-block INT8 arithmetic is performed in the systolic array. FP32 arithmetic is performed across blocks.

Fig. 4.2 depicts the connection between high-level modules of the MXITA architecture. For intra-block INT8 arithmetic, MXITA employs a 4D systolic array for INT8 MAC operations. The integer outputs of this systolic array are subsequently converted to FP32 and scaled with the MX block scales by the normalizer. These resulting scaled FP32 values are accumulated and finally forwarded to the output dataflow combiner.

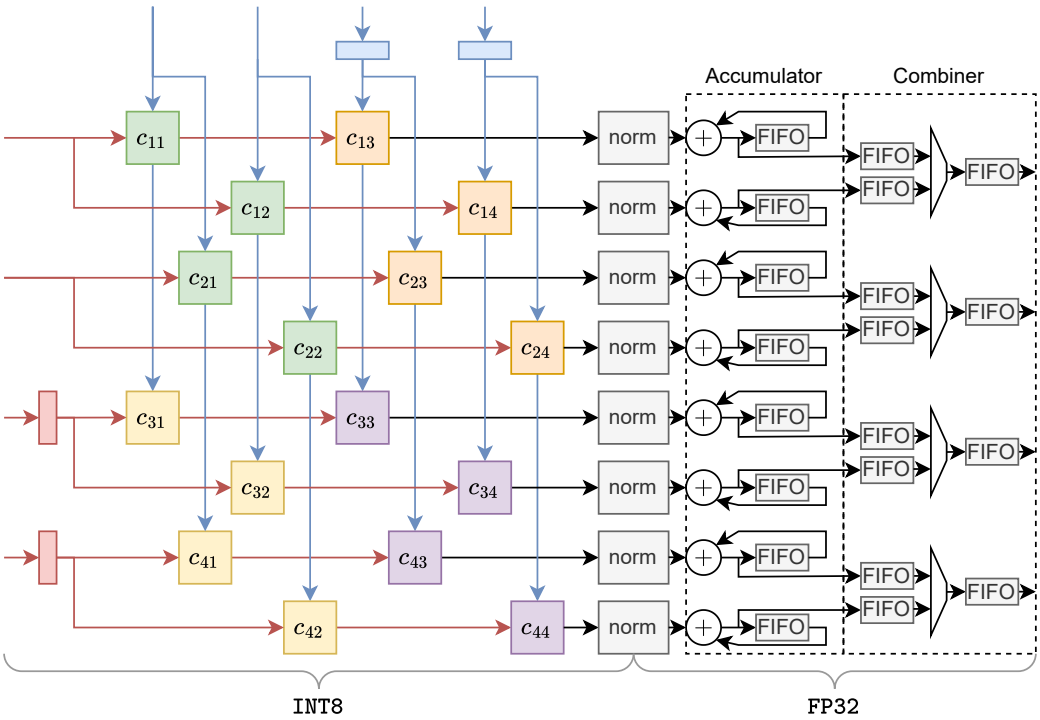


Figure 4.2: Complete MXITA architecture with 4D systolic array, normalizers, accumulators and combiner

4.1. 4D Systolic Array

The primary motivation for adopting a 4D systolic array design instead of a conventional 2D systolic array lies in the time-sharing of the inter-block FP32 arithmetic units. In a 2D systolic array, the time required to fully empty the systolic array relies on its row length M, requiring M cycles to propagate its output. For an output-stationary PE as illustrated in Fig. 2.6, while intra-block INT8 MAC operations are performed, an MX dot-product result for an MX block size k is produced every k cycles. This introduces the constraint $k \geq M$, which in turn limits the feasible size and scalability of the 2D systolic array to the minimum supported MX block size, denoted as k_{min}. By contrast, the 4D systolic array used in MXITA (Fig. 4.3) avoids this limitation by ”partitioning” the conventional 2D systolic array into a 2D grid of smaller 2D systolic arrays. With this approach, the systolic row length M can be reduced to support smaller MX block sizes, while additional parameters (N, P, Q) allow for scaling up the 4D systolic array to achieve a comparable throughput to conventional 2D systolic arrays.

The 4-dimensional systolic array, shown in Fig. 4.3, is parameterized by four dimensions (M, N, P, Q). The parameters M and N specify the number of sequential stages for the activation and weight matrices, respectively, while P and Q define the degree of broadcasting for the activation and weight matrices. In this design, the shift registers for de-skewing the systolic output matrix are omitted (as opposed to the 4D systolic array design presented in Fig. 2.13), thereby reducing the amount of sequential logic required for implementing the 4D systolic array. In addition to the 4D systolic array, a finite state machine (FSM) manages operand transactions by counting up to the MX block size k for each transaction. After k transactions have been issued to the systolic array, the accumulated results are propagated (or “popped”) using the same mechanism described in Fig. 2.7.

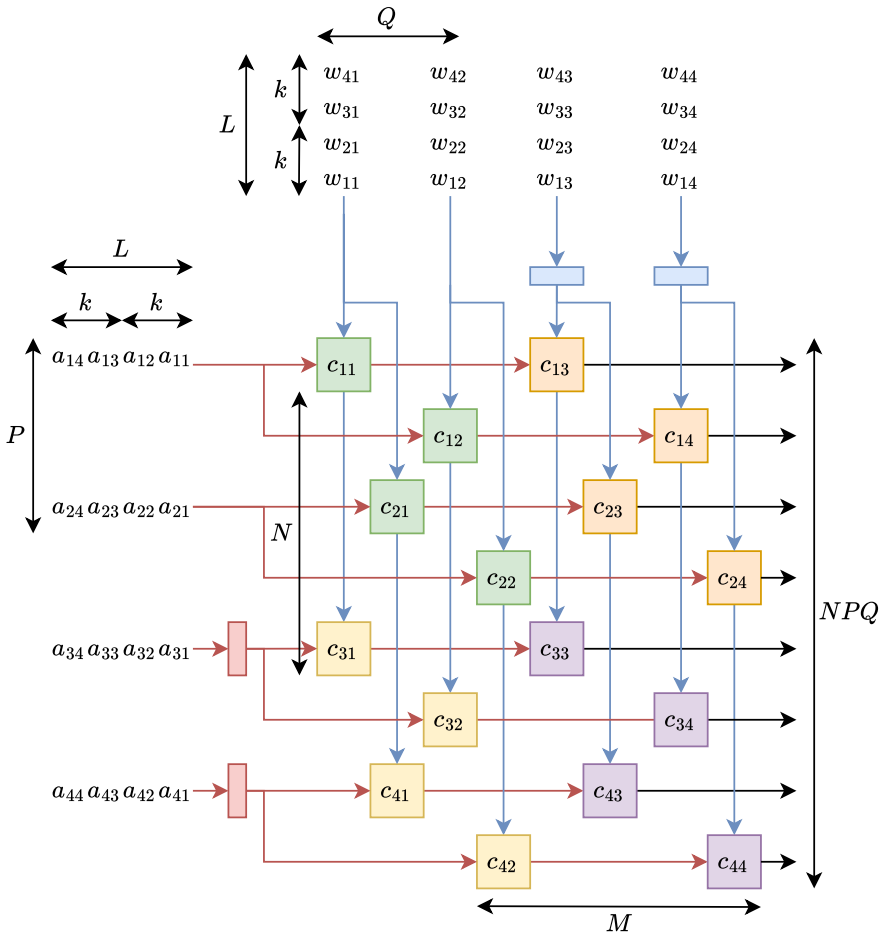


Figure 4.3: 4D systolic array design with output-stationary PEs in configuration $(M, N, P, Q) = (2, 2, 2, 2)$. Depicted is an MX block size $k = 2$ and an inner dimension length $L = 4$ as input matrix multiplication operands

An alternative view of the 4D systolic array is shown in Figure 4.4, where it can be interpreted as a 2D array of MN nodes, each containing PQ PEs. These nodes form PE “islands” or ”nodes”, with each node producing a PQ-sized sub-matrix per cycle (highlighted in Fig. 4.3). This functional representation emphasizes a key advantage of the 4D design: reduced sequential logic for propagating input and weight matrices. In Fig. 4.3, each PE has a flip-flop after the operand broadcast, but equivalently, a flip-flop can be placed before the broadcast as in Fig. 4.4 to further minimize the amount of sequential logic.

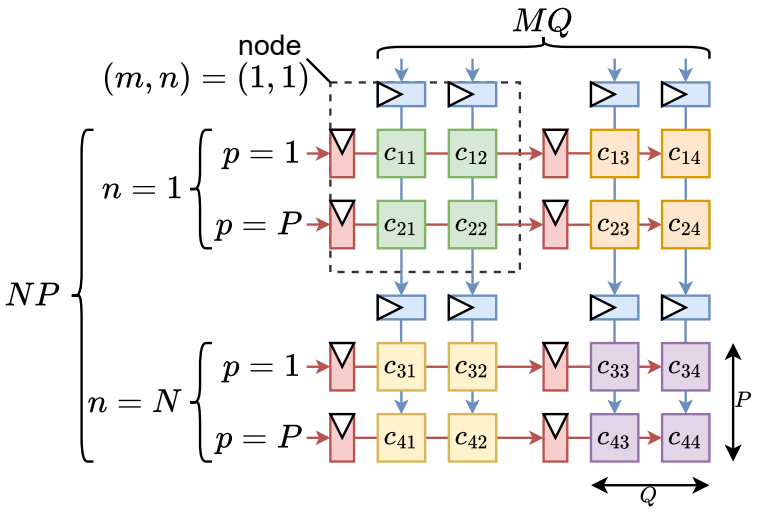


Figure 4.4: Functional representation of the 4D systolic array

With the functional representation shown in Fig. 4.4, the temporal order in which the nodes produce output results (as illustrated in Fig. 4.5) becomes more apparent. The temporal order of which node generates an output results is identical to a 2D systolic array, except that each node produces a PQ sub-matrix instead of a single output element.

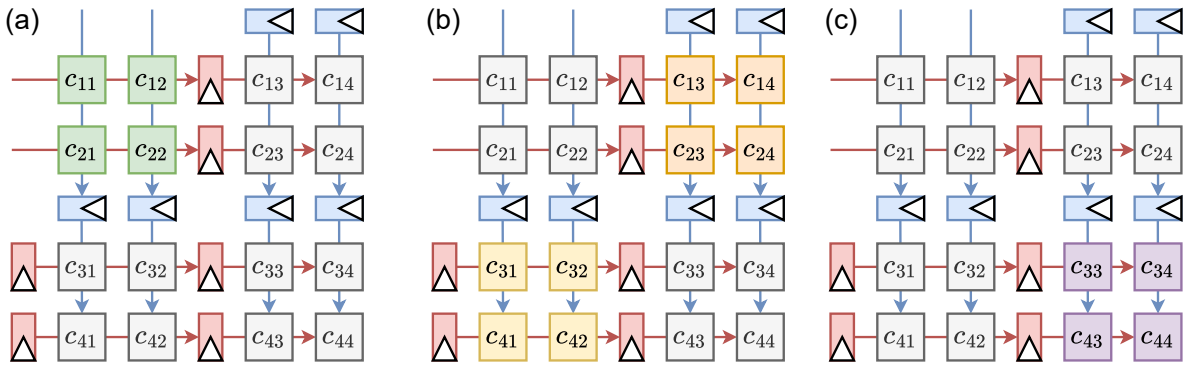


Figure 4.5: Temporal order of 4-D systolic array output generation

4.2. Normalizer

The normalizer unit is responsible for dequantizing the results produced by MX arithmetic, ensuring that outputs can be accumulated in a consistent data format (i.e. FP32) across different MX blocks. As illustrated in Figure 4.6, this process involves first casting the integer results generated by the systolic array into an FP32 representation. The corresponding block scale is then applied by adding its value to the FP exponent, such that the dequantized value of the partial output matrix sum is represented in FP32.

The main complexity of the normalizer architecture stems from the manner in which MX scales are mapped onto the output results of the 4D systolic array. As illustrated in Fig. 4.7, the output matrix element c_{rc} in a given row r and column c corresponds directly to scale values in the same row of the input matrix S_r and the same column of the weight matrix S_c. For instance, the output element c_{wy}, located at row w

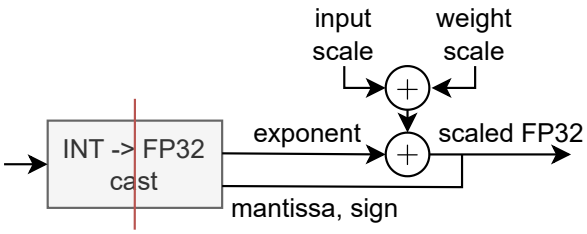


Figure 4.6: Normalizer unit, containing a pipelined INT to FP32 cast unit, and 8-bit adders for exponentiating the FP32 output with MX scales

and column y, is computed from row w of the input matrix and column y of the weight matrix. Consequently, this element must be normalized using the scales associated with row w of the input and column y of the weight matrix.

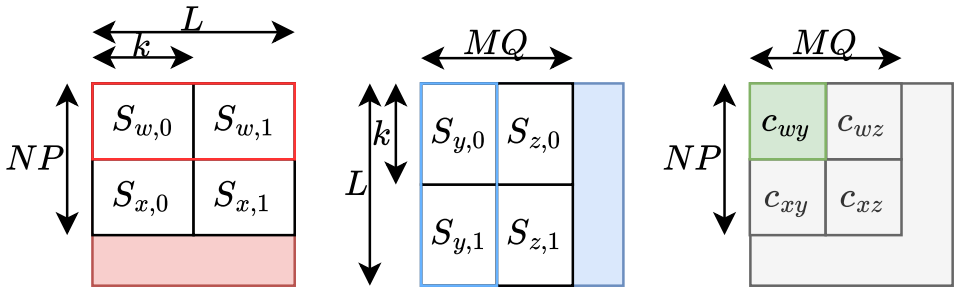


Figure 4.7: Distribution of MX input (red) and weight (blue) scales over output (green) matrix elements, for a 2×2 output matrix

In Fig. 4.8, the connections are illustrated for a single row of "nodes" $n = 1$ of the 4D systolic array. The input of each normalizer unit is time-multiplexed across M elements of the systolic array. As all output matrix elements c_{rc} within the same output matrix row r or column c share the same input or weight scale respectively, these scales are broadcasted to the corresponding normalizer units. On top of this spatial dataflow of input and weight scale distribution, the weight scale distribution consists of an additional temporal dataflow. For example, in Fig. 4.8, weight scale FIFO 1 is matched to columns $1, 3$. As these columns are time multiplexed to the normalizer input, new weight scales need to be retrieved every cycle. Consequently, the weight scale FIFOs are popped for each cycle when the systolic array produces an output, while the input scale FIFOs are popped once every k cycles for an MX block size k.

As Fig. 4.8 only illustrates the normalizer dataflow for $N = 1$, scale reuse can also extend along the N dimension, as shown in Fig. 4.9. Weight scales are propagated down the columns, since PEs in subsequent rows r may share the same column c. Because each row in the N dimension produces outputs one cycle later, additional flip-flops are inserted to correctly skew the weight scales to each island row. These flip-flops also manage the timing of the pop signal for the input scale FIFOs, ensuring that each PE receives the appropriate scales at the correct cycle.

Finally, the input to the normalizer units from the systolic array will be stalled if any of the following conditions occur when the systolic array is about to produce an output:

- The input scale FIFOs are empty.
- Any weight scale FIFOs contain fewer than M scales.
- The output of the normalizer is blocked.

These conditions ensure that the normalizer always has the required scales and space to process the systolic array outputs correctly.

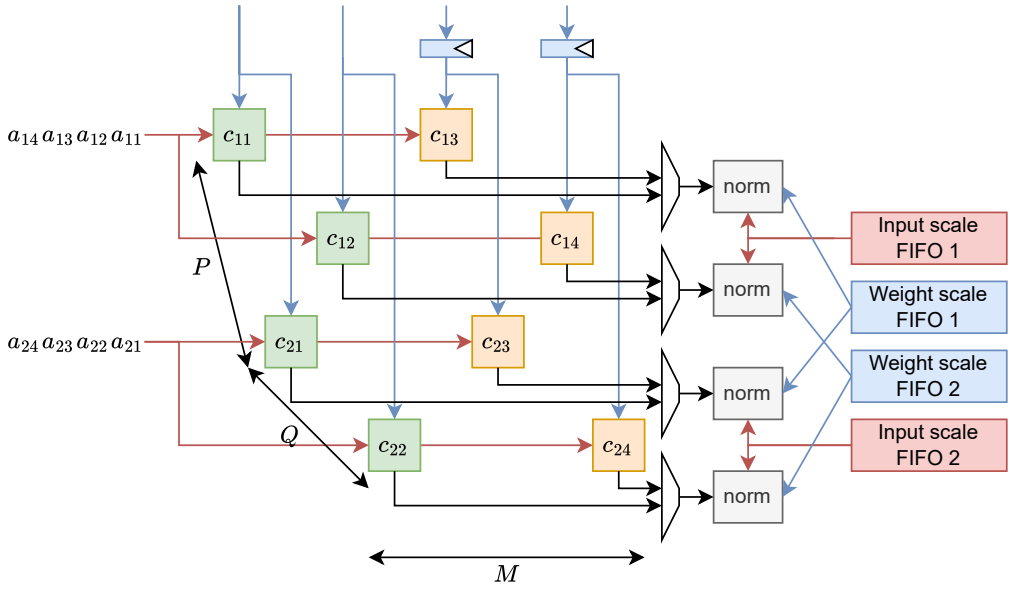


Figure 4.8: Close-up diagram for the connections of the normalizer units for a 4D systolic array with dimensions $(M, N, P, Q) = (2, 1, 2, 2)$ and input/weight scale FIFOs

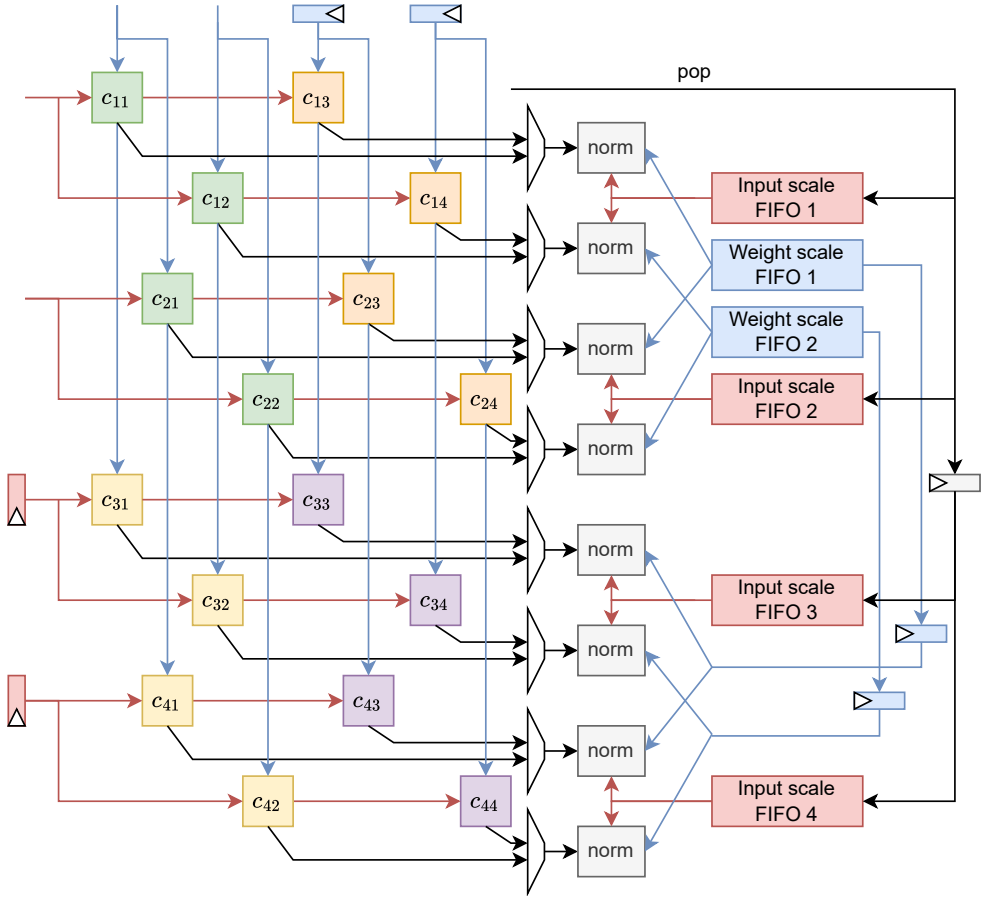


Figure 4.9: Diagram for input and weight scale sharing over the 4D systolic array output with dimensions $(M, N, P, Q) = (2, 2, 2, 2)$

4.3. Accumulator

The purpose of the accumulator is to perform inter-block accumulation of the partial sums generated by a MX dot product. This accumulation is carried out for each element of the output matrix. Fig. 4.10 illustrates the accumulator architecture for a single systolic array row and its connection to the normalizer unit, as this module is replicated across all NPQ systolic array rows. Each accumulator handles a single output element per cycle, while storing M partial sums in a looped FIFO buffer.

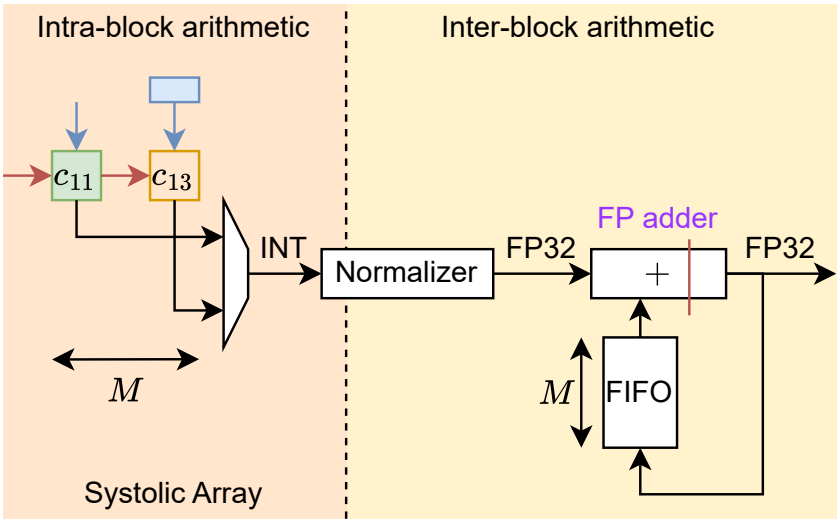


Figure 4.10: Accumulator architecture for 4D systolic row $(n, p, q) = (1, 1, 1)$. This design only works when the FP adder has less than M pipeline registers

When the dot product result of the first block along the inner dimension arrives, the output accumulator FIFO is initially empty. In this case, the second FP adder operand is 0, and the FIFO is not popped. The resulting FP32 sums are pushed into the output accumulator FIFO if the current block is not the last block. On the final block, the FIFO is emptied as its FIFO output data is fed into the second operand of the FP32 adder.

4.4. Output combiner

The output combiner illustrated in Fig. 4.11 reorganizes the 4D systolic array output matrix into a 2D output matrix by time-multiplexing NPQ FIFOs of M deep over the Q dimension. This process reduces the number of required output ports from NPQ to NP. Furthermore, the output combiner removes the skew inherent in the systolic array output such that whole columns of the output matrix can be written. At the peak throughput, the output combiner receives $MNPQ$ FP32 elements every L_{min} cycles, and transmits NP elements each cycle. Therefore, $NP \geq \frac{MNPQ}{L_{\text{min}}}$ has to be valid to match this throughput without any stalls.

The diagram in Fig. 4.11 illustrates the dataflow for a $(M, N, P, Q) = (2, 2, 2, 2)$ 4D systolic array, which produces the entire output matrix within three cycles. (a) The first PE island generates PQ elements. (b) As the next two PE islands produce their outputs in a systolic manner, all output ports of the combiner have valid data, and elements $c_{11}, c_{21}, c_{31}, c_{41}$ are popped. (c) Simultaneously, the last PE island's outputs are pushed into their corresponding FIFOs.

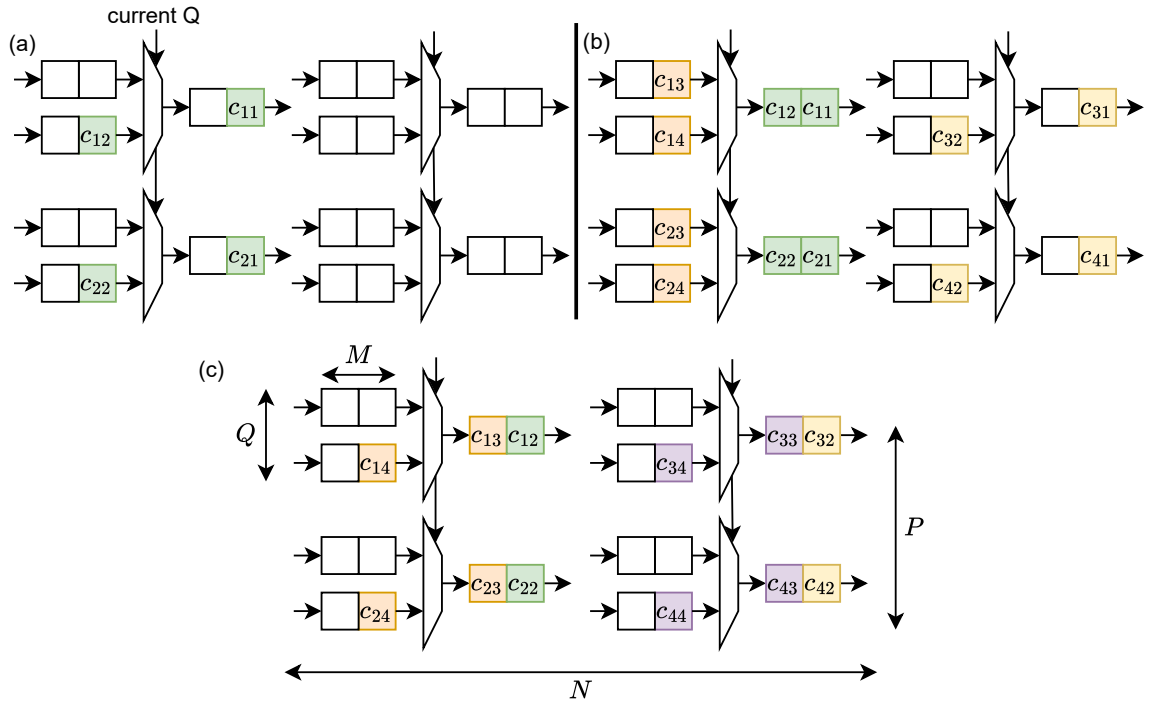


Figure 4.11: Output combiner design and dataflow for $(M, N, P, Q) = (2, 2, 2, 2)$

4.5. Hardware Processing Engine (HWPE) integration

The MXITA accelerator is integrated with the a multi-processor cluster, as illustrated in Figure 4.12. This processor cluster consists of several single-cycle RISC-V ”Snitch” cores, and a L1 Tightly Coupled Data Memory (TCDM) functioning as an accelerator scratchpad. Configuration of the MXITA accelerator is performed via the narrow AXI interconnect, where the MX block size k and matrix inner dimension L are set. Additionally, the memory addresses of the input and weight matrices, as well as the corresponding scales, are provided to the HWPE control module. Once configured, execution is initiated by a Snitch core. During execution, the HWPE independently accesses the TCDM through the wide TCDM interconnect. Upon completion, the HWPE signals the Snitch cluster via an interrupt.

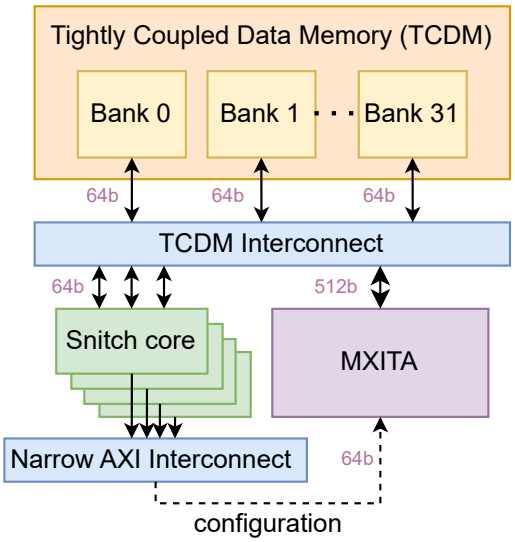


Figure 4.12: Snitch cluster, MXITA accelerator, wide TCDM interconnect for data transfer and narrow AXI interconnect for configuration

The HWPE integration is illustrated in Figure 4.13. A HWPE consists of

- The HWPE engine (MXITA accelerator)
- HWPE fence for fencing different dataflows
- FIFOs and data buffers
- HWPE streamer containing sources and sinks for initiating TCDM data transfers
- Multiplexer for multiplexing the different dataflows
- HWPE control module for configuration of the HWPE streamer and engine

The HWPE is parameterized to initiate 512-bit wide transactions with the TCDM. This data width is chosen based on the required average accelerator bandwidth, which will be discussed in detail in Chapter 5. Within the HWPE streamer, sources and sinks initiate TCDM read and write transactions, respectively. The HWPE multiplexer arbitrates which source or sink can initiate the next transaction. For a read transaction, the returned data is routed from the multiplexer and streamer source to the corresponding data buffer. As each accelerator data operand may not consume the entire 512-bit packet in a single transaction, the data element in the HWPE buffers are processed over multiple transactions. To ensure that both the activation and weight matrix data are available simultaneously for the systolic array, a HWPE fence is used, guaranteeing that both data packets can be transacted together.

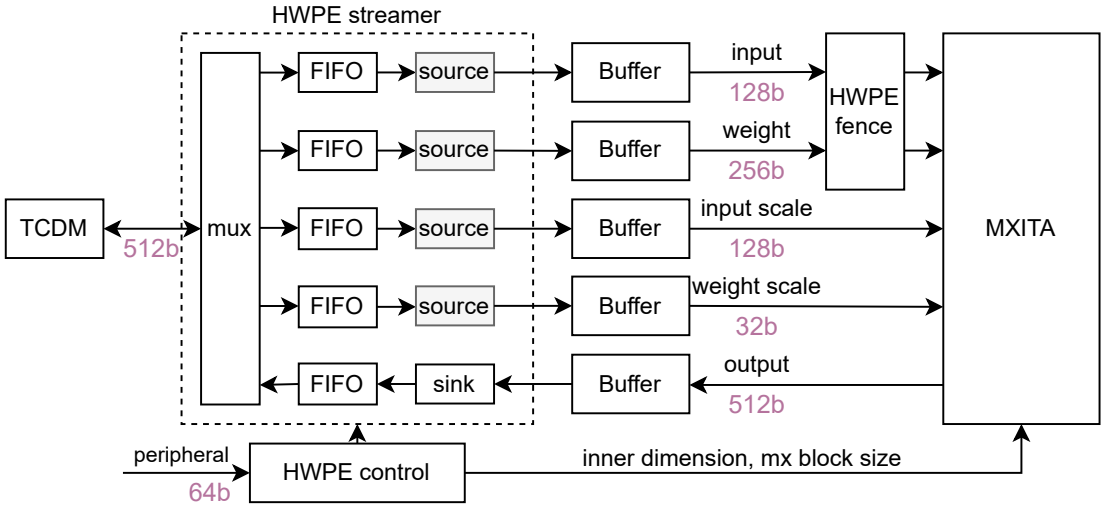


Figure 4.13: HWPE integration for MXITA with parameters $(M, N, P, Q) = (8, 4, 4, 4)$

An example of TCDM transactions for all MXITA operands is shown in Fig. 4.14. Each TCDM transaction for the corresponding data operand (input / weight matrix, or scales) has a 512-bit width. This width is chosen to match the total average accelerator bandwidth, which is larger than the amount of data that each operand of the MXITA accelerator consumes per cycle. Consequently, the MXITA accelerator can take multiple transactions per operand to consume an entire 512-bit TCDM packet. For instance, the input matrix operand requires $8NP = 128$ bits per transaction. Therefore, a 512-bit packet can be consumed over $\frac{512}{8NP} = 4$ transactions. In cases where the MXITA accelerator requires fewer than 512 bits in total for a given MX matrix multiplication operand (e.g., the input scale in Figure 4.14), the remaining data elements in the packet are padded with zeros.

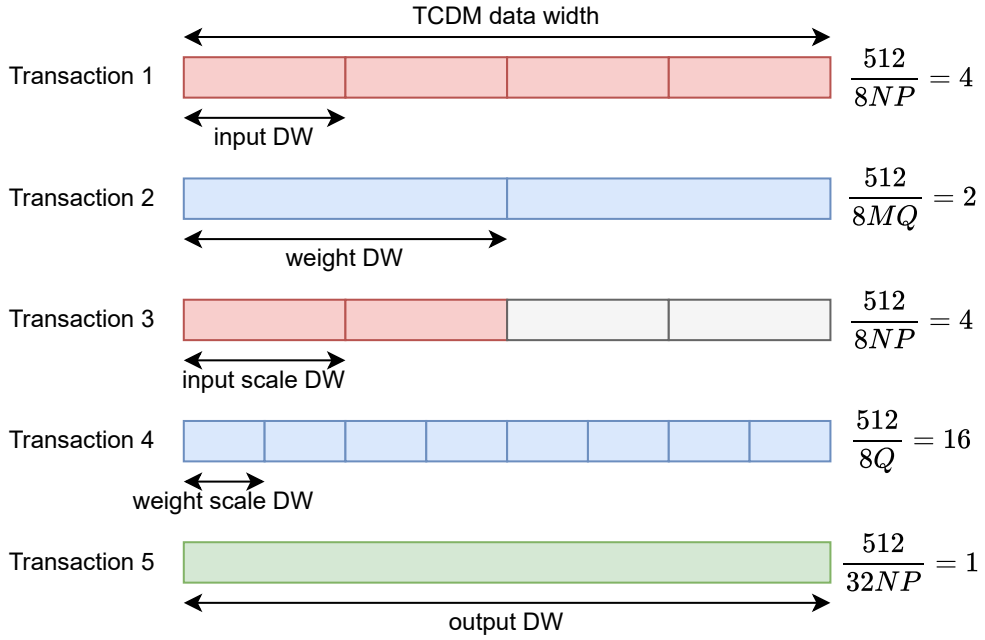


Figure 4.14: Example 512b TCDM transactions for $(M, N, P, Q) = (8, 4, 4, 4)$

5

Design Implementation

This chapter presents an overview of the targeted performance requirements, which form the basis for the chosen parameterization of the MXITA architecture. These performance requirements span over not only the desired accelerator performance and supported workloads, but also the execution of entire neural network workloads at the system-level. Subsequently, the functional verification methodology of the MXITA architecture is described.

5.1. Performance analysis

5.1.1. Accelerator parameterization

The required throughput and the minimum supported block size k_{min} jointly constrain the design space of MXITA, directly determining the choice of the design parameters (M, N, P, Q). These four parameters define both the size of the 4D systolic array and the number of instantiated inter-block arithmetic modules, thereby influencing not only the performance but also the area and timing characteristics of the design. To ensure that MXITA sustains the desired performance while maximizing reuse of expensive FP32 intra-block arithmetic hardware, the design parameters are subject to several constraints. Among these, the most critical are the minimum block size $k_{\text{min}} > M$, which sets a lower bound on supported MX block sizes k while maximizing the reuse factor M, and the required throughput, quantified as $MNPQ$, which dictates the overall performance target in MACs/cycle. In turn, these design parameters influence the resulting required memory bandwidth of each accelerator data operand:

- Input bandwidth $8NP$ bits/cycle
- Weight bandwidth $8MQ$ bits/cycle
- Input scale bandwidth $8\frac{NP}{k}$ bits/cycle
- Weight bandwidth $8\frac{MQ}{k}$ bits/cycle
- Output bandwidth $32\frac{MNPQ}{L_{\text{min}}}$ bits/cycle

The accelerator is designed to support a minimum block size of $k_{\text{min}} = 8$ and a peak throughput of $MNPQ = 512$ MAC operations per cycle. To satisfy these constraints, the design fixes $M = 8$ and allows a set of possible combinations for (N, P, Q). Among these, the configuration $(M = 8, N = 4, P = 4, Q = 4)$ was selected as the baseline configuration for MXITA. It should be noted that, as synthesis does not account for physical routing effects, the accelerator area scales approximately linearly with the parameters (M, N, P, Q). Consequently, this introduces a direct trade-off between the reuse factor M of the output accumulator and the flexibility of the minimum block size k_{min}, since increasing M improves FP32 hardware reuse but simultaneously restricts the supported MX block size k.

5.1.2. System-level execution

To support complete application workloads, the HWPE is integrated into the Snitch cluster, enabling post-processing of the output matrix results. Such post-processing may include activation functions, which typically require multiple cycles to execute on a Snitch core. Since the output throughput of MXITA depends on the inner dimension L, namely $\frac{MNPQ}{L}$ FP32 elements per cycle, it is necessary to select a minimum inner dimension, L_{dim}, that ensures the system can sustain full throughput under worst-case conditions.

To determine an appropriate value for L_{dim} and evaluate the performance requirements imposed on the Snitch cluster, we analyze the DeiT-Ti [15] transformer workload as a representative case study.

The Snitch cores are responsible for several key computations. In particular, they execute the softmax operation on $Q \times K^\intercal$, as well as the layer normalization applied to both the MHA output and the feed-forward network. We assume that each Snitch core requires approximately 2 cycles to compute the softmax of a single element. Due to the data dependencies in the softmax operation, output matrix must first be transferred to the L1 TCDM before the softmax operation can begin. In MXITA, the softmax is computed row-wise over the $Q \times K^\intercal$ output matrix.

Figure 5.1 illustrates the computation timeline, showing the matrix multiplications accelerated on MXITA and the operations executed on the Snitch cores (highlighted in orange). The softmax computation can commence as soon as a set of rows of the $Q \times K^\intercal$ matrix becomes available, and can overlap with subsequent matrix multiplications such as $A \times V$. This overlap is enabled by the tiling of the operand matrices: for instance, the $A \times V$ multiplication can be launched before the entire A matrix is computed, since only the first NP rows of A are required to calculate the first NP rows of $A \times V$.

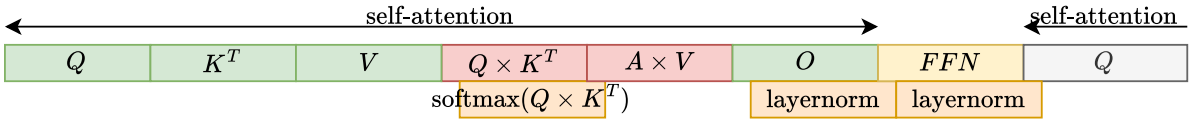


Figure 5.1: Computational flow for the transformer multi-head attention layer as depicted in Figure 2.15, where $A = \text{softmax}(Q \times K^\intercal)$ and FFN represents the feed-forward network

By reordering the sequence of matrix multiplications, as shown in Figure 5.2, the latency constraint for computing softmax$(Q \times K^\intercal)$ can be further relaxed. Specifically, inserting the computation of V between $Q \times K^\intercal$ and $A \times V$ enables the softmax evaluation to be overlapped with the independent computation of V. This scheduling effectively hides part of the softmax latency on the Snitch cores, providing additional slack and allowing more cycles for its execution without impacting the overall execution time of the multi-head attention computation.

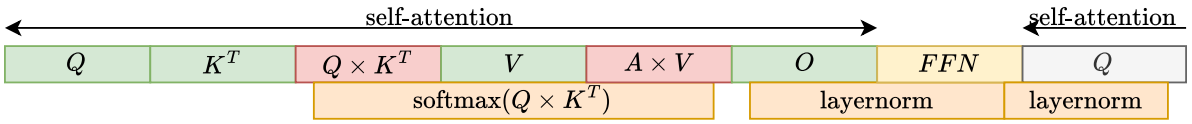


Figure 5.2: Alternative computational flow for the multi-head attention layer

While the performance requirements of the Snitch cores can be drastically relaxed with efficient reordering of computations, MXITA is designed to be a general MX matrix multiplication accelerator for applications where such patterns might not occur. Therefore, assuming a (worst case scenario of) minimum inner dimension size $L_{\text{min}} \geq 128$, an output throughput of $512/64 = 4$ FP32 elements / cycle can therefore be matched with the 2 cycles / FP32 element throughput per Snitch core, giving 8 Snitch cores per Snitch cluster.

While efficient reordering of computations can significantly relax the performance requirements of the Snitch cores, MXITA is designed as a general-purpose MX matrix multiplication accelerator, targeting applications where such optimizations may not be applicable. Therefore, under a conservative worst-case assumption with a minimum inner dimension size of $L_{\text{min}} \geq 128$, the accelerator achieves an output throughput of $\frac{512}{128} = 4$ FP32 elements per cycle. This throughput can be sustained by provisioning 8 Snitch cores per cluster, as each Snitch core provides a throughput of 1 FP32 element every 2 cycles.

5.2. Verification

5.2.1. Functional

The verification of the MXITA accelerator was conducted at three hierarchical levels: the MXITA accelerator toplevel, the HWPE integration, and the Snitch cluster integration. For the accelerator toplevel, a Python + SystemVerilog co-simulation framework was developed, as illustrated in Figure 5.3. In this setup, the Python golden reference generates random 8-bit input matrices, weight matrices, and scale values, which are then passed to the SystemVerilog testbench as stimulus for the Design Under Test (DUT). The DUT produces an output matrix file containing FP32 values, which is subsequently numerically compared against the FP32 golden reference.

Direct bitwise comparison of FP32 values was avoided, as both the golden reference and the DUT perform floating-point computations according to slightly different implementations of the FP32 specification. Instead, a numerical tolerance criterion of $|\text{REF} - \text{DUT}| < 0.0001\%$ was adopted. Within this margin, all test results matched. However, it was observed that the numerical error between the golden reference and the DUT increases with larger inner dimensions L, due to the accumulation of a greater number of errors within the FP32 operations.

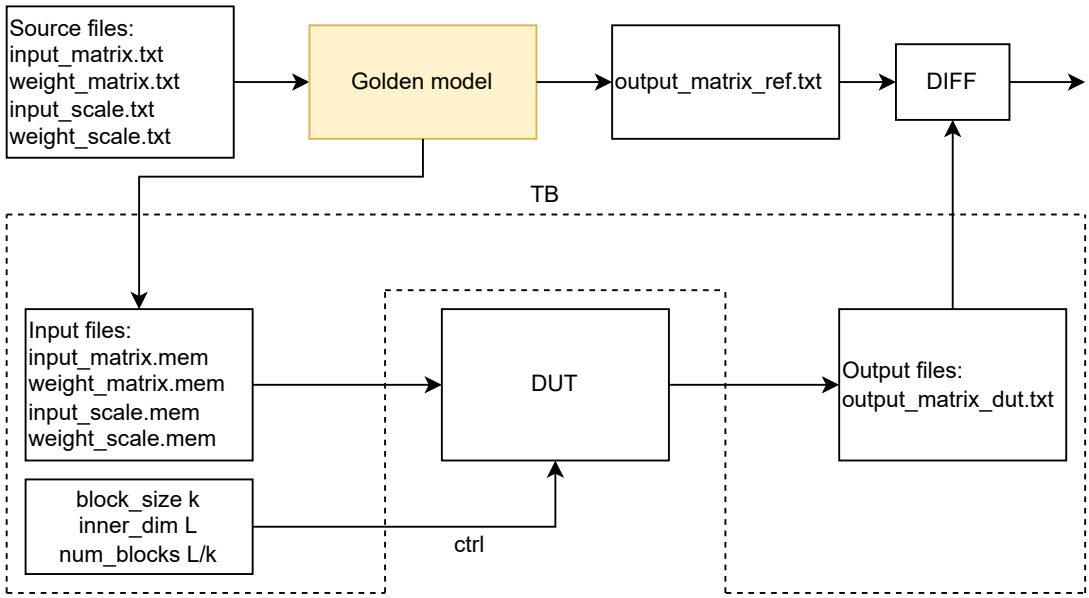


Figure 5.3: Functional verification setup

5.2.2. Design for Testability (DFT)

Automated Testpattern Generation

The distribution of the input and output values presented to the DUT is a key characteristic when analyzing whether the design has been tested under realistic conditions. In practice, the outputs of neural network layers often exhibit non-uniform distributions, typically centered around zero and characterized by varying degrees of sparsity. Ensuring that the accelerator produces output distributions consistent with expected statistical behavior is therefore crucial, as it validates not only the correctness of the numerical computations but also the representativeness of the testbench stimuli. This, in turn, provides confidence that the design will behave robustly across real workloads rather than being overfitted to synthetic or overly simplistic test vectors.

In the verification setup, the 8-bit input, weight, and scale matrices are sampled from a uniform distribution in order to ensure wide and unbiased coverage of the representable value space. This sampling strategy provides a balanced evaluation scenario that does not inherently favor specific ranges of values. After performing the matrix multiplications and accumulations within MXITA, the resulting output matrix exhibits a normal (Gaussian-like) distribution centered around zero. This behavior is expected as a consequence of the central limit theorem, since the summation of many independent uniformly distributed random variables asymptotically converges toward a normal distribution.

6 Results

In this chapter, synthesis results of the MXITA accelerator for different configurations will be presented. Then, physical implementation results of the HWPE integrated accelerator with the Snitch cluster will be presented. Finally, these results will then be compared with Performance Power Area (PPA) metrics of other state-of-the-art works. However, due to limited time, power results from the physical implementation could not be retrieved.

6.1. Synthesis

The MXITA design was synthesized for different (M, N, P, Q) parameterizations using Synopsys DC with register retiming enabled, targeting the GlobalFoundries 22nm technology node (SS corner, $125°$C, 0.72 V). As a first step, the synthesis results were analyzed for the baseline configuration $(M, N, P, Q) = (8, 4, 4, 4)$. The corresponding toplevel accelerator area and timing characteristics are noteed in Table 6.1, with the detailed area breakdown illustrated in Fig. 6.1.

Fig. 6.2 shows an area-time plot of the $(M, N, P, Q) = (8, 4, 4, 4)$ configuration across the achieved clock frequencies, ranging from 1334 MHz down to 400 MHz. The design points form a Pareto front, illustrating the trade-off between area and timing. From this curve, a target frequency of 800 MHz is recommended, as it lies near the marginal rate of substitution. The marginal rate of substitution represents the point on the Pareto front where improving one metric (e.g., frequency) starts to incur disproportionately higher costs in the other metric (e.g., area). Choosing a target frequency at this point ensures a balanced trade-off, achieving high performance without unnecessarily increasing the design area.

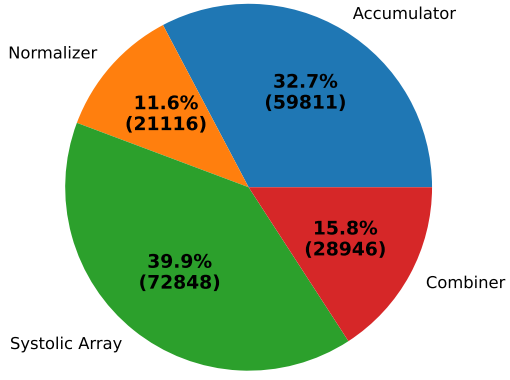


Figure 6.1: Area breakdown (in μm^2) of the MXITA accelerator for $(M, N, P, Q) = (8, 4, 4, 4)$

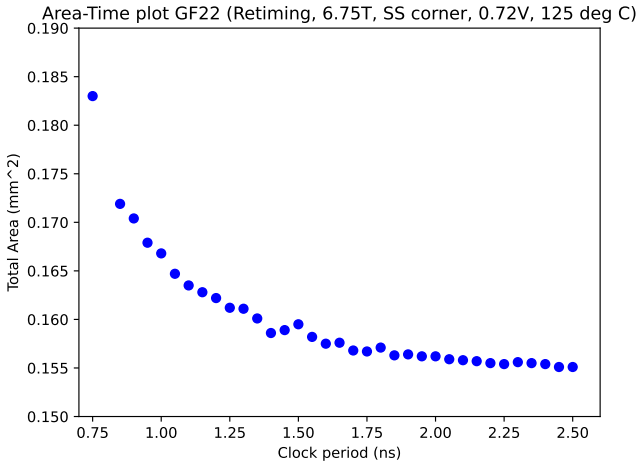


Figure 6.2: Area-time plot for $(M, N, P, Q) = (8, 4, 4, 4)$ and different target clock frequencies

Table 6.1: Synthesis results of the MXITA accelerator for different configurations, showing the trade-off between peak MAC throughput, area, and timing across varying M, N, P, Q parameters.

M	N	P	Q	Peak perf. (MACs/cycle)	Area (mm^2)	Timing (MHz)
8	4	4	4	512	0.183	1334
16	4	4	2	512	0.158	1363
32	4	2	2	512	0.147	1383
32	16	1	1	512	0.161	1380
32	1	4	4	512	0.150	1375
8	8	4	4	1024	0.365	1282
8	4	8	4	1024	0.352	1278
8	4	4	8	1024	0.350	1276
16	4	4	4	1024	0.301	1304
32	4	4	2	1024	0.278	1346
8	8	4	8	2048	0.698	1219
16	8	4	4	2048	0.593	1250
32	4	4	4	2048	0.544	1296

Next, the design size was increased to investigate scalability of the throughput in terms of area and timing. Table 6.1 shows that scaling the design along either M or (N, P, Q) affects the total accelerator area differently. Increasing M has a smaller impact on total area, as it primarily determines the degree of inter-block peripheral reuse. Maximizing reuse is desirable as the FP32 inter-block arithmetic hardware occupy more area than the systolic array itself.

However, M also sets the minimum block size k_{min} that can be supported without stalling the systolic array, which in turn limits by how much M can be increased. Regarding timing, the critical path lies within the 1-stage pipelined floating-point unit and appears to be limited to 1.3 GHz. Furthermore, with the output multiplexer chain as illustrated in Fig. 2.7, the critical path shifts to the 4D systolic array at $M \geq 32$, causing a drop in maximum frequency from approximately 1300 MHz to 900 MHz. Therefore, the optimized output selection design illustrated in Fig. 2.8 is implemented, maintaining a maximum ≈ 1.3 GHz clock frequency across the entire parameter space.

To summarize, varying (M, N, P, Q) will have the following design trade-offs: Increasing M improves inter-block reuse and reduces area growth, but at the cost of larger minimum MX block sizes. Scaling (N, P, Q) increases throughput but directly increases area without providing peripheral reuse benefits.

6.2. Physical Implementation

After analysis of the design tradeoffs for different systolic array parameters, we perform the physical implementation of the Snitch cluster and HWPE integrated MXITA accelerator in the $(M, N, P, Q) = (8, 4, 4, 4)$ configuration. Fig. 6.3 shows the synthesis area breakdown of the Snitch cluster. It can be seen that the additional area overhead from the HWPE integration can account for 36% of the HWPE total area.

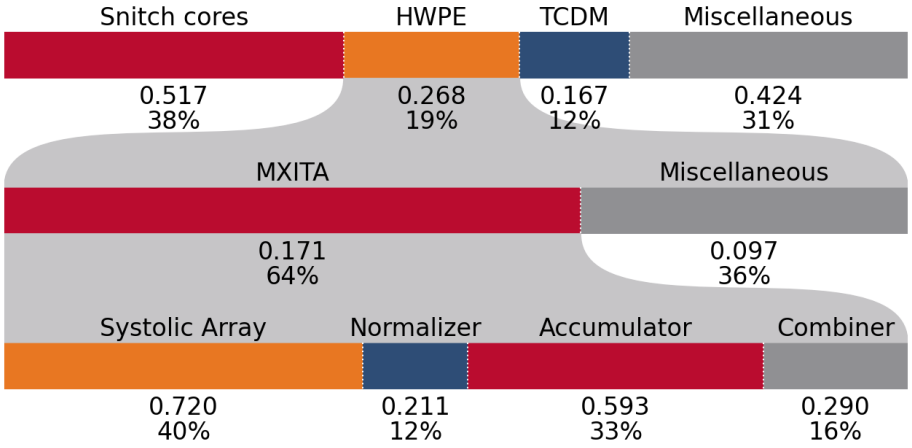


Figure 6.3: Area breakdown of Snitch cluster in mm^2

Fig. 6.4 and Fig. 6.5 shows the physical implementation of the entire Snitch cluster with a 128 kB TCDM and the 4D systolic array respectively. An overall target frequency of 800 MHz was achieved. Furthermore, it was found that the physical implementation area of the MXITA accelerator is approximately the same as area results reported after synthesis. In the future, further analysis of the 4D systolic array scalability should be performed with larger configurations (1024, 2048+ MACs/cycle).

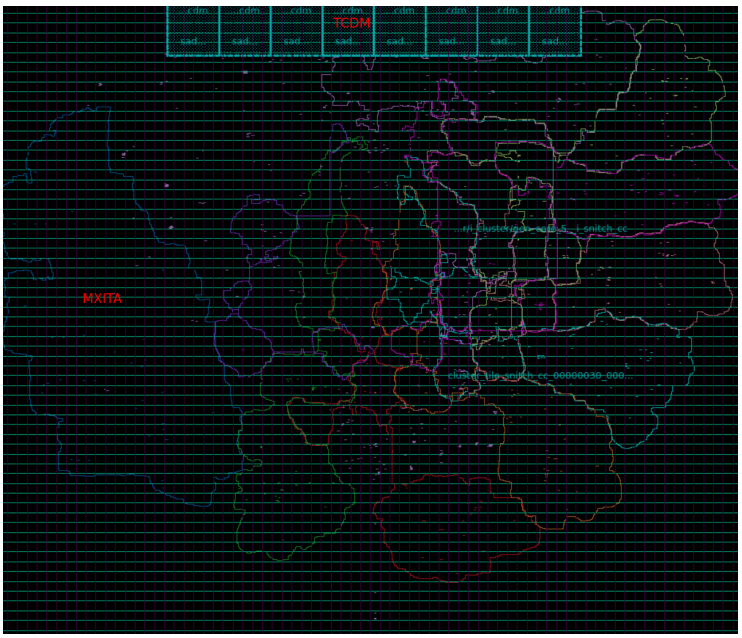


Figure 6.4: Physical implementation of Snitch cluster with MXITA in configuration $(M, N, P, Q) = (8, 4, 4, 4)$ over a 2×2 mm^2 floorplan area

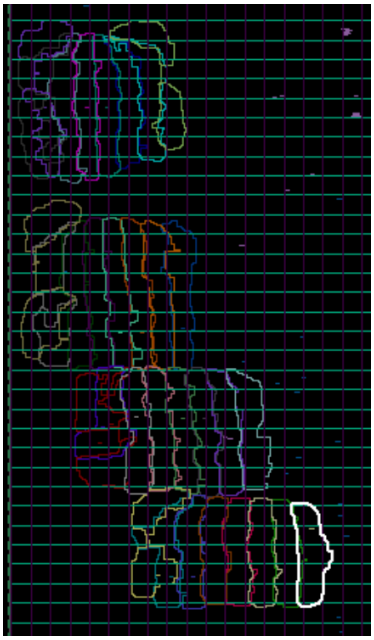


Figure 6.5: Placement of 4D systolic array with configuration $(M, N, P, Q) = (8, 4, 4, 4)$. Each group contains $PQ = 16$ INT8 MAC units

Table 6.2: Comparison of state-of-the-art accelerators and the proposed MANTA. Power results for MXITA are not yet obtained.

Metric	HSA [10]	Cuyckens *et al.* [11]*	JackUnit [12]*	This work*
Tech. [nm]	28	16	65	22
Dataflow	MMM, MVM	MMM	MMM	MMM
Area [mm^2]	0.287	1.064	12.04	0.16
Frequency [MHz]	500	400	400	800
Power [W]	0.0864	3.98 (MXINT8)	1.853	—
Data format	INT8 (MMM), MXINT4 (MVM)	MXFP4/6/8, MXINT8	bloat16, FP8, INT4/8, MXFP8, MXINT4/8	MXINT8
Peak Perf. [TOPS]	0.512 (MMM), 0.128 (MVM)	3.28	0.819 (MXINT8)	0.819
Area Eff. [TOPS/mm^2]	1.784 (MMM), 0.446 (MVM)	3.08	0.068 (MXINT8)	5.12 (MXINT8)
Energy Eff. [TOPS/W]	5.93 (MMM), 1.48 (MVM)	0.823 (MXINT8)	0.442 (MXINT8)	—

* Synthesis results

6.3. Comparison

MXITA will be quantitavely compared with HSA [10], Cuyckens et al. [11], and Jack Unit [12]. Since these works report PPA metrics under different conditions (technology nodes, synthesis vs. post-layout, and varying throughput definitions), we normalize their results for a fair comparison.

Performance

The performance in TOPS can be derived from the clock frequency and the number of MACs performed per cycle. We count a MAC operation to be 2 operations (multiplication, addition). For MXITA, 512 PEs supporting 512 MACs/cycle and a 800 MHz clock frequency allows the design to reach $2 \times 512 \times 0.8 \times 10^9/10^{12} = 0.819$ TOPS. HSA [10] has implemented 256 PEs operating at 500 MHz, giving a peak performance of 0.512 TOPS. However, during the HSA's MVM dataflow, the PE utilization drops to 25% resulting in a throughput of 0.128 TOPS. Cuyckens et al. [11] implement 4096 PEs at 400 MHz, which leads to a peak performance of $\frac{2\times4096\times400\times10^6}{10^{12}} = 3.277$ TOPS. Similar calculations were done for the performance results of Jack Unit [12], where 1024 PEs with full utilization at 400 MHz leads to a peak performance of $\frac{2\times1024\times400\times10^6}{10^{12}} = 0.819$ TOPS.

Area

For the area, we only take the PE area for a fair comparison, as in this work we focus on the accelerator implementation instead of its system-level integration. In HSA [10], the PEs take up 44.5% of the reported 0.646 mm^2 total post-layout design area in TSMC 28nm CMOS technology, for an effective 0.287 mm^2 PE area. Cuyckens et al. [11] report a MAC synthesis area of 2078.42 μm in TSMC 16nm FinFET technology. By extrapolating the single PE area to their 4096 implemented PEs, we obtain a systolic array area of 8.511 mm^2 out of the total 8.92 mm^2 design area. Jack Unit [12] presents a synthesis area of 11762 μm^2 per MAC unit in a 65 nm CMOS library, giving a 12.04 mm^2 PE area for their 32×32 implemented PEs. For MXITA, the results are obtained from synthesis of the entire accelerator using GF22 technology. However, due to the accelerators being implemented in different process technologies, a normalized area quantity i.e. Gate Equivalent needs to be determined for a more fair comparison.

Power

For the energy efficiency, we take the reported energy consumption of MAC operations from HSA [10], the energy consumption per operation per PE from Cuyckens et al. [11] and the average power consumption per PE from Jack Unit [12]. However, as of the moment of writing this thesis, power results of the MXITA accelerator are not yet obtained due to blocking issues in the physical design implementation flow.

Nevertheless, in Table 6.2 it can be seen that MXITA outperforms both HSA [10], Cuyckens et al. [11] and Jack Unit [12] in terms of area efficiency, due to MXITA lacking flexiblity of supporting different dataflows and/or support for the entire MX specification. Furthermore, part of the area efficiency discrepancy can be attributed to the lack of area normalization across different process technologies.

7

Conclusion

The motivation behind this work stems from the rapid growth of deep learning models, particularly Transformers, which has far outpaced hardware scaling. While INT8 quantization reduces memory footprint, it often degrades accuracy. The MXINT8 format mitigates this trade-off by grouping INT8 values with a shared exponent, preserving FP32-level accuracy while achieving up to $4\times$ memory savings in Transformer workloads. However, exploiting these formats requires specialized hardware capable of efficiently handling mixed integer–floating-point operations. Prior systolic MX accelerators have been limited by underutilized processing elements or costly FP32 peripheries.

This thesis presented MXITA, a multi-dimensional systolic array accelerator for efficient execution of Microscaling (MX) matrix multiplications in neural network workloads. The architecture was designed, implemented, and verified, with integration into the Snitch cluster demonstrating both functional correctness and system-level compatibility. The parameterizable design, defined by (M, N, P, Q), enables trade-offs between the minimum supported MX block size and the degree of FP32 peripheral reuse, while maintaining consistent throughput. Post-processing performance requirements of the Snitch cores were also analyzed to ensure balanced system-level operation.

Verification confirmed correctness across accelerator internals, HWPE integration, and Snitch co-processing. Synthesis in GF22 technology quantified the impact of (M, N, P, Q) scaling on area, timing, and critical paths, identifying optimal configurations along the area–time Pareto frontier. Results showed that MXITA achieves higher area efficiency than prior state-of-the-art accelerators by amortizing the cost of FP32 hardware across compute tiles, thereby reducing overhead compared to designs supporting multiple dataflows or the full MX specification.

8

Future work

TCDM bank conflict mitigation
The L1 TCDM, with which the HWPE is integrated, consists of a memory bank structure, as (simplified) illustrated in Figure 8.1. For clarity, the MX scales are omitted from the diagram. Each bank in the L1 TCDM supports only a single 32-bit read or write per cycle. Therefore, if multiple ports attempt to access elements within the same bank simultaneously, a bank conflict occurs, causing memory accesses to stall and thereby reduces accelerator performance. To prevent these conflicts, the operands of the MX matrix multiplication should be arranged in memory so that all simultaneous accesses target different banks. Access patterns differ between the input and weight matrices, as shown in Figure 2.13: input matrix tiles are accessed column-wise, whereas weight matrix tiles are accessed row-wise. Future work could implement a mechanism, either in hardware or software, to reorganize operands for more efficient memory utilization.

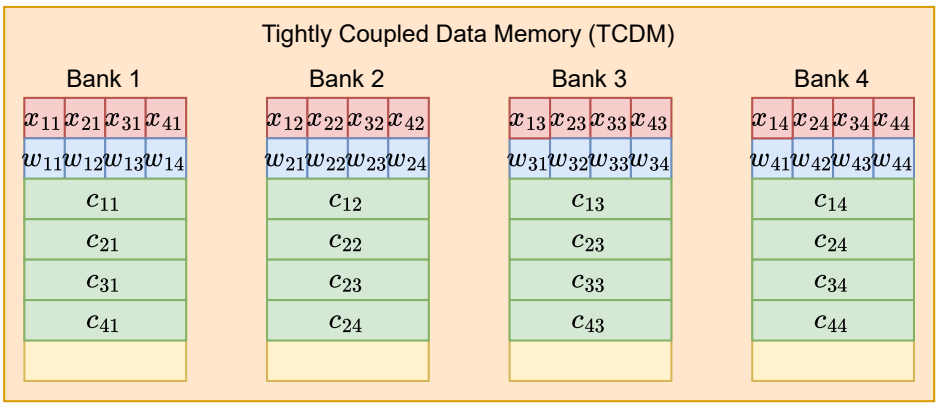


Figure 8.1: TCDM architecture and memory organization of input and output matrix operands

Output matrix transpose
A potential hardware approach for reorganizing the output matrix operand is to transpose the resulting output matrix within the MXITA dataflow combiner module. The combiner architecture supporting the output transpose is illustrated in Figure 8.2. Notably, this architecture is identical to the non-transposed combiner shown in Figure 4.11, with the only differences being the FIFO dimensions and multiplexer selection. Consequently, MXITA could be configured with $M = N$ and $P = Q$ so that the output matrix can be transposed with minimal architectural changes and negligible area overhead in future implementations.

Output matrix quantization
Transferring the FP32 output matrix to the TCDM allows the Snitch cores to directly perform post-processing operations on the output matrix. However, this approach demands high data bandwidth due to the size of FP32 elements. To reduce this bandwidth requirement, additional circuitry can be integrated within the output dataflow combiner to perform MX quantization before writing the result back to the L1 TCDM. As discussed in Section 2.1, the maximum reasonable MX block size for balancing neural network inference accuracy and memory savings is $k = 32$. Moreover, MX re-quantization only requires identifying the largest

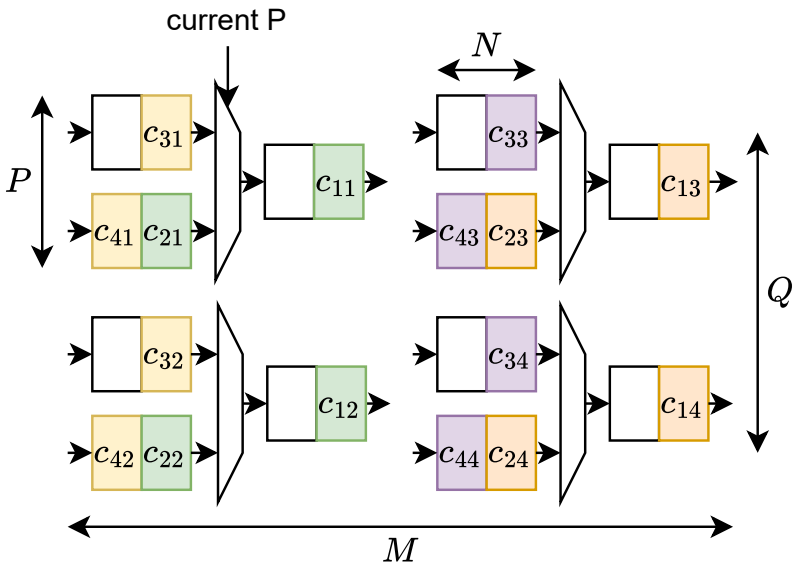


Figure 8.2: Output dataflow combiner architecture for transposing the output operand matrix of the 4D systolic array

power-of-two smaller than the maximum of k elements. Therefore, the optional FIFOs shown in Figure 4.11 can be sized to hold k elements per row, and a divider or re-quantization module can be placed at the output of the dataflow combiner module.

Multiple HWPE execution contexts
Configuring and initiating execution of the HWPE accelerator incurs a latency overhead that becomes significant for small inner dimensions L. To mitigate this overhead caused by the Snitch cluster software interface and the HWPE, the configuration and execution of the HWPE can be overlapped by implementing multiple control contexts within the HWPE integration.

L1 TCDM matrix tiling and reuse
Any computations performed on the L1 TCDM require the data to be first loaded from the L2 memory. However, the L2 memory bandwidth is limited, and in systems with multiple Snitch clusters, this limited bandwidth must be shared among the clusters. Therefore, reusing tiles loaded into the L1 TCDM reduces the number of L2 memory accesses, at the cost of increased total L1 TCDM memory consumption, by reusing tile operands as illustrated in Figure 8.3.

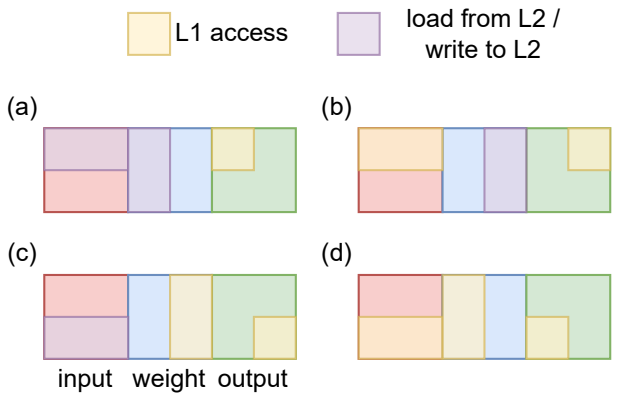


Figure 8.3: L1 TCDM output stationary operand reuse for two input and weight tile operands each.

References

[1] Jared Kaplan et al. *Scaling Laws for Neural Language Models*. 2020. arXiv: 2001.08361 [cs.LG]. URL: https://arxiv.org/abs/2001.08361.

[2] R.R. Schaller. "Moore's law: past, present and future". In: *IEEE Spectrum* 34.6 (1997), pp. 52–59. DOI: 10.1109/6.591665.

[3] Marian Verhelst et al. "How to Keep Pushing ML Accelerator Performance? Know Your Rooflines!" In: *IEEE Journal of Solid-State Circuits* 60.6 (2025), pp. 1888–1905. DOI: 10.1109/JSSC.2025.3553765.

[4] Norman P. Jouppi et al. *In-Datacenter Performance Analysis of a Tensor Processing Unit*. 2017. arXiv: 1704.04760 [cs.AR]. URL: https://arxiv.org/abs/1704.04760.

[5] Bita Darvish Rouhani et al. *Microscaling Data Formats for Deep Learning*. 2023. arXiv: 2310.10537 [cs.LG]. URL: https://arxiv.org/abs/2310.10537.

[6] H. T. Kung. "Why Systolic Architectures?" In: *Computer* 15.1 (Jan. 1982), pp. 37–46. DOI: 10.1109/MC.1982.1653825. URL: https://doi-org.tudelft.idm.oclc.org/10.1109/MC.1982.1653825.

[7] Brian Chmiel et al. *FP4 All the Way: Fully Quantized Training of LLMs*. 2025. arXiv: 2505.19115 [cs.LG]. URL: https://arxiv.org/abs/2505.19115.

[8] Ashish Vaswani et al. *Attention Is All You Need*. 2023. arXiv: 1706.03762 [cs.CL]. URL: https://arxiv.org/abs/1706.03762.

[9] Gamze Islamoglu et al. "ITA: An Energy-Efficient Attention and Softmax Accelerator for Quantized Transformers". In: *2023 IEEE/ACM International Symposium on Low Power Electronics and Design (ISLPED)*. IEEE, Aug. 2023, pp. 1–6. DOI: 10.1109/islped58423.2023.10244348. URL: http://dx.doi.org/10.1109/ISLPED58423.2023.10244348.

[10] Chun-Ting Chen et al. *Hybrid Systolic Array Accelerator with Optimized Dataflow for Edge Large Language Model Inference*. 2025. arXiv: 2507.09010 [cs.AR]. URL: https://arxiv.org/abs/2507.09010.

[11] Stef Cuyckens et al. *Efficient Precision-Scalable Hardware for Microscaling (MX) Processing in Robotics Learning*. 2025. arXiv: 2505.22404 [cs.AR]. URL: https://arxiv.org/abs/2505.22404.

[12] Seock-Hwan Noh et al. *Jack Unit: An Area- and Energy-Efficient Multiply-Accumulate (MAC) Unit Supporting Diverse Data Formats*. 2025. arXiv: 2507.04772 [cs.AR]. URL: https://arxiv.org/abs/2507.04772.

[13] Norman P. Jouppi et al. "A domain-specific supercomputer for training deep neural networks". In: *Commun. ACM* 63.7 (June 2020), pp. 67–78. DOI: 10.1145/3360307. URL: https://doi-org.tudelft.idm.oclc.org/10.1145/3360307.

[14] Yuqi Xue et al. "V10: Hardware-Assisted NPU Multi-tenancy for Improved Resource Utilization and Fairness". In: *Proceedings of the 50th Annual International Symposium on Computer Architecture*. ISCA '23. Orlando, FL, USA: Association for Computing Machinery, 2023. DOI: 10.1145/3579371.3589059. URL: https://doi.org/10.1145/3579371.3589059.

[15] Hugo Touvron et al. *Training data-efficient image transformers & distillation through attention*. 2021. arXiv: 2012.12877 [cs.CV]. URL: https://arxiv.org/abs/2012.12877.