Full-Stack TPU Implementation and Integration of FPGA vs. ASIC

Anthony Hermez[1] and Vivek Keval[2]

[1, 2]*Chandra Family Department of Electrical and Computer Engineering*
[1, 2]*The University of Texas at Austin*

[1]anthony.hermez@utexas.edu
[2]vivek.keval@utexas.edu

***Abstract*— Tensor Processing Units (TPUs) and related architectures (such as Tensor Cores) have become a necessary component of hardware compute for machine learning and high-performance workloads due to their performant and power-efficient execution of matrix multiplication, convolution, and non-linear activation functions. Google's TPU is an evolving ASIC that leads the market in tensor computation. However, Google's performance analysis is limited to only a high-level overview of the TPU architecture, with minimal discussion on microarchitecture and software interfacing. Moreover, the TPU design is only analyzed on an ASIC implementation. We propose a Google-inspired TPU design and a custom compiler to interface and approximate realistic machine learning workload execution times. We do not claim our design to be more performant than Google; however, we seek to expose the underlying microarchitecture of the TPU, memory interface challenges, and hardware-software codesign choices necessary in the TPU design. Moreover, we compare our design on FPGA and ASIC implementations and compare PPA metrics (power, performance, and area/resource utilization) for the respective designs.**

***Keywords*— machine learning hardware, hardware/software codesign, compiler, microarchitecture, performance analysis**

I. Introduction

The growing demand for efficient computation in machine learning and high-performance workloads has driven the development of specialized hardware accelerators, with Tensor Processing Units (TPUs) emerging as a leading solution. TPUs, along with architectures such as NVIDIA's Tensor Cores, achieve significant gains in performance and power efficiency by optimizing the execution of matrix multiplication, convolution operations, and non-linear activation functions — the foundational operations in modern neural networks. Among these, Google's TPU series represents the forefront of ASIC-based tensor computation.

Despite their prominence, detailed public information regarding Google's TPU microarchitecture, memory hierarchy, and software interface remains scarce. Existing analyses primarily focus on high-level architectural descriptions, leaving a gap in understanding the specific design choices and trade-offs involved in TPU development. Furthermore, Google's performance evaluations of TPUs are specifically limited to ASIC implementations, without exploration of alternative platforms such as FPGAs that could offer additional insights into the design space.

In this work, we present a Google-inspired TPU microarchitecture along with a custom compiler designed to bridge software and hardware for realistic machine learning workloads. Our objective is not to outperform Google's designs, but rather to illuminate the microarchitectural considerations, memory interfacing challenges, and hardware-software codesign strategies necessary in constructing a TPU. Additionally, we implement and evaluate our design on both FPGA and ASIC platforms, providing a comprehensive comparison of power, performance, and area (PPA) metrics across the two implementations.

II. Related Works

This section reviews prior work on systolic array architectures and the dataflow models commonly employed in tensor processing units (TPUs). We discuss three principal dataflow strategies: Weight Stationary, Output Stationary, and Row Stationary. These dataflows influence how data is scheduled and reused across the array, significantly affecting performance and energy efficiency. Furthermore, we examine related architectures such as Google's TPU, which served as a conceptual foundation for our design, and NVIDIA's Tensor Cores. Finally, we highlight the key distinctions between our implementation and existing solutions

A. Systolic Architectures for Matrix Multiplication

Systolic arrays are a class of highly parallel architectures well-suited for general matrix multiplication (GEMM) operations that form the computation backbone of deep neural networks (DNNs). These architectures consist of a grid of

tightly-coupled processing elements (PEs) that perform Multiply-and-Accumulate (MAC) operations in a pipelined fashion. The defining characteristic of a systolic array is its data movement pattern where data flows rhythmically through the array, with each PE passing intermediate results to its neighbor. This regular structure eliminates the need for complex control logic, global interconnects, or memory address generation, yielding substantial area and energy efficiency benefits.

In a typical implementation, input activations and weights are loaded into the array from orthogonal directions and intermediate results are accumulated locally within the PEs and eventually forwarded out, usually through the bottom edge. The regular and deterministic movement of data allows for significant reuse of input data, weights, and partial sums, which is crucial for performance and reducing memory bandwidth requirements. Systolic arrays are employed in a wide range of accelerators including Google's TPU [1], which exemplifies a scale-up approach, and Microsoft's Brainwave platform [2], which demonstrates a scale-out strategy.

The effectiveness of these architectures, however, depends heavily on the underlying dataflow and workload. Dataflow determines how operands are reused, how intermediate results are stored, and how memory bandwidth is consumed. Several common strategies have emerged, each with distinct trade-offs in reuse efficiency and performance. The following subsections will briefly cover three different dataflows: *Weight Stationary*, *Output Stationary* and *Row Stationary*.

1. *Weight Stationary Dataflow:*

The weight stationary (WS) dataflow is a mapping strategy in systolic arrays that aims to maximize the reuse of weight values by keeping them fixed within the PEs for the duration of computation. Before execution begins, the weights (i.e., elements of convolutional filters or matrices) are pre-loaded into the PEs and remain stationary throughout the computation. The input feature map (IFMAP) elements are then streamed in from the left edge of the array and multiplied with the stationary weights, with each PE generating partial sums which are reduced across rows to form the final output.This dataflow is particularly effective when filter reuse is high, such as when multiple convolution windows apply the same filter across different input regions. The spatial mapping assigns one filter column-wise across the array and processes multiple convolution windows in parallel across rows, while the temporal dimension corresponds to the depth of the convolution window. As a result, WS dataflow reduces the need for frequent reloading of weights from SRAM, which is advantageous in scenarios where off-chip memory access is a bottleneck. However, this benefit comes with trade-offs in utilization and array scheduling complexity. For instance, WS dataflow may underutilize compute resources when the number of filters or output feature map pixels is not an exact multiple of the array size.

2. *Output Stationary Dataflow:*

The output stationary (OS) dataflow focuses on maximizing reuse of partial sums by fixing the output pixel in each PE for the entire duration of its computation. In this approach, each PE is responsible for computing a single output pixel, accumulating partial results across multiple cycles. Both the IFMAP and weight data are streamed into the array and propagated through the array using nearest-neighbor links.

OS dataflow is well-suited for convolutional layers in DNNs, where each output pixel is formed as the dot product of a convolution window and a filter. Since output pixels are stationary in the PEs, intermediate values do not need to be written back to SRAM until the computation is complete. This approach significantly reduces write bandwidth and energy cost associated with memory accesses. Despite its advantages in write efficiency and partial sum reuse, OS dataflow can lead to increased complexity in data input scheduling and potentially lower utilization if the number of output pixels per layer does not align well with the array dimensions.

3. *Row-Stationary Dataflow:*

The row-stationary (RS) dataflow is a hybrid mapping strategy introduced in the Eyeriss accelerator design [3]. It aims to simultaneously exploit reuse of all three operand tensors: weights, inputs, and partial sums. RS achieves this by keeping a small subset of input rows and filter weights stationary in PEs across a given row, while the convolution windows are slid across columns. By carefully organizing the data layout and inter-PE communication, the RS dataflow ensures that each PE can reuse inputs across multiple filters and reuse weights across multiple input activations, all while accumulating partial sums locally. This balance makes it highly effective for convolutional layers, particularly in scenarios where there is spatial locality in both input and filter data. Compared to WS and OS dataflows, RS achieves higher data reuse and often better energy efficiency because it minimizes redundant data movement across all memory levels. However, the implementation of RS is more complex due to the need to coordinate reuse of all operands across both dimensions of the array.

B. *Existing TPU Architectures*

In developing our architecture, we studied existing tensor processing architectures, primarily Google’s TPU and NVIDIA’s Tensor Cores. The Google TPU [1], introduced as

a domain-specific ASIC for accelerating neural network inference, is built around a systolic array optimized for 8-bit integer matrix multiplication. It is highly efficient at executing deep learning operations with low power consumption and high throughput, especially in large-scale data center environments. The TPU architecture integrates components such as unified buffers, activation units, and high-bandwidth memory interfaces to support complex neural network workloads with minimal latency.

NVIDIA's Tensor Cores [4], on the other hand, are integrated within their general-purpose GPUs and are designed to accelerate mixed-precision matrix operations like fused multiply-add (FMA). Present in GPUs such as the A100, Tensor Cores support data types like FP16, BF16, and TF32, and are well-suited for both training and inference tasks. Their integration with CUDA cores and large memory bandwidth enables them to handle both AI-specific and general-purpose computation within the same architecture. These designs represent state-of-the-art solutions for high-performance AI computation and served as important references for our work.

C. *Our TPU relative to existing designs*

While inspired by the Google TPU, our implementation is a significantly simplified version tailored specifically for 8-bit inference tasks. Like the TPU, our architecture centers around a systolic array, but it supports only unsigned 8-bit matrix multiplication. It does not include support for matrix addition, activation functions, or other computational units found in the Google TPU. Our goal was to replicate the core functionality of the TPU's matrix multiplication engine without the added complexity, creating a minimal, working proof-of-concept design.

In terms of I/O and memory, our system is much more limited. Unlike the Google TPU's high-throughput interface and unified buffer system, our implementation features a much more basic I/O mechanism, which constrains performance and scalability. Additionally, compared to NVIDIA's highly integrated and versatile Tensor Cores, our design is narrowly focused and operates as a standalone inference accelerator without support for training or mixed-precision operations.

Due to the scarcity of detailed public documentation on commercial TPU internals, our approach emphasized understanding architectural principles over matching commercial performance. The project was conceived with educational intent: to develop a simplified and functional tensor processing unit that maintains structural fidelity while remaining small enough to be implemented and tested within limited hardware constraints.

III. TPU Architecture

This section will cover the key microarchitectural components of our TPU implementation. Figure 1 illustrates the block-level view of the TPU.

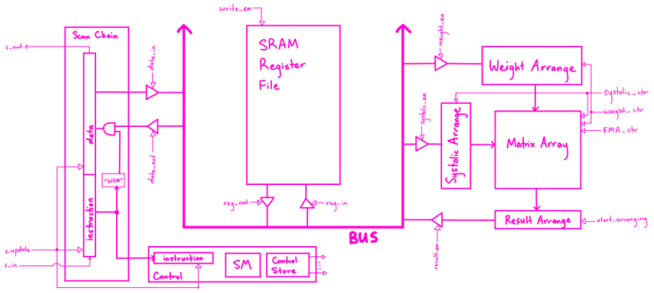

Figure 1. TPU Block-Level View.

A. ISA

The TPU uses a simple 5-instruction CISC ISA to operate. Table 1 lists these instructions, their execution times (in cycles), and a description of what they do. Note that N refers to the dimension of the square systolic array.

Opcode	Exec. Time	Description
RHM	3	Reads host memory into a TPU register.
LW	N+3	Loads TPU register into the Weight Arrange block.
LS	3	Loads TPU register into the Systolic Arrange block.
MM	3N+1	Multiplies the Weight and Systolic blocks and stores result in a TPU register.
WHM	3	Writes a TPU register to host memory.

Table 1. TPU ISA.

B. Control Unit

The TPU control uses a microcoded state machine and control signals. Figure 2 illustrates the state diagram for the 5 instructions. This helps derive the execution times from Table 1.

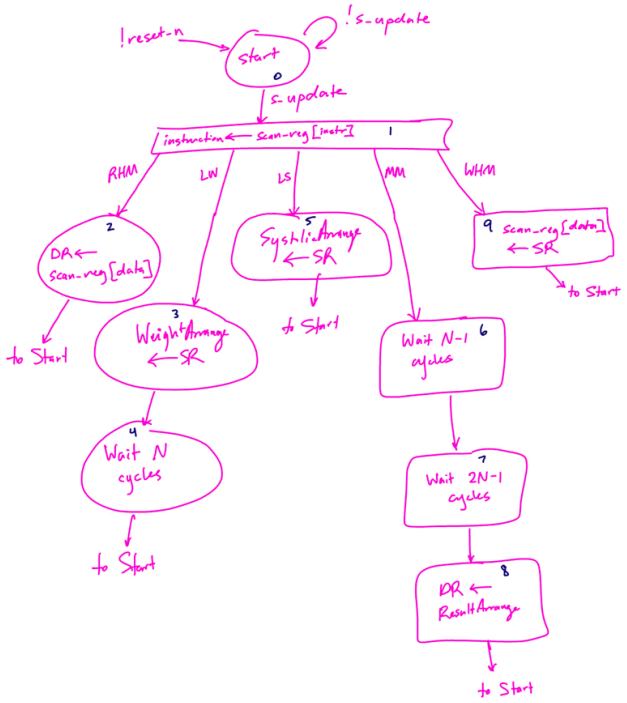

Figure 2. TPU State Diagram.

There are 12 microcoded control signals for the TPU design described in Figure 3 below.

Description of state	State ID	data_out	data_in	reg_out	reg_in	write_en	weight_en	weight_ctr	systolic_en	systolic_ctr	FMA_ctr	result_en	start_arranging
Start	0000 (state 0)	0	0	0	0	0	0	0	0	0	0	0	0
instruction <- scan_reg[inst] (decode)	0001 (state 1)	0	0	0	0	0	0	0	0	0	0	0	0
DR <- scan_reg[data]	0010 (state 2)	0	1	0	1	1	0	0	0	0	0	0	0
WeightArrange <- SR	0011 (state 3)	0	0	1	0	0	1	0	0	0	0	0	0
Wait N Cycles (weight forwarding)	0100 (state 4)	0	0	0	0	0	0	1	0	0	0	0	0
SystolicArrange <- SR	0101 (state 5)	0	0	1	0	0	0	0	1	0	0	0	0
Wait N-1 Cycles (systolic forwarding)	0110 (state 6)	0	0	0	0	0	0	0	0	1	1	0	0
Wait 2N-1 Cycles (arrange results)	0111 (state 7)	0	0	0	0	0	0	0	0	1	1	0	1
DR <- ResultArrange	1000 (state 8)	0	0	0	1	1	0	0	0	0	0	1	0
scan_reg[data] <- SR	1001 (state 9)	1	0	1	0	0	0	0	0	0	0	0	0

Figure 3. TPU Microcode.

C. *MatrixArray*

Similar to the Google TPU, our TPU design uses a weight-stationary systolic array of fused-multiply-add (FMA) processing elements (PEs) to perform the matrix multiplication operation. Figure 4 illustrates the architecture of the processing elements. More specifically, the MatrixArray performs the operation C = AB, where A, B, C ∈ $\{NxN\ matrices\ of\ 8bit\ integers\}$. Using this notation, our implementation maps A as "weight inputs" and B as "systolic inputs." The weights must be preloaded into the Matrix Array before the systolic inputs propagate in. The next section discusses how data is arranged going into and out of the Matrix Array.

D. *Data Arrangement*

The TPU has three units for data management: *WeightArrange*, *SystolicArrange*, and *ResultArrange*. Each arrange block solves a unique problem (Figure XXX). The *WeightArrange* block takes an input matrix and maps it to N transposed row vectors. The *SystolicArrange* block maps an input matrix to 2N-1 column vectors with a rhomboidal data pattern. The *ResultArrange* block solves the opposite problem of the *SystolicArrange* block and maps 2N-1 row vectors with rhomboidal data patterns and maps them to a single result matrix.

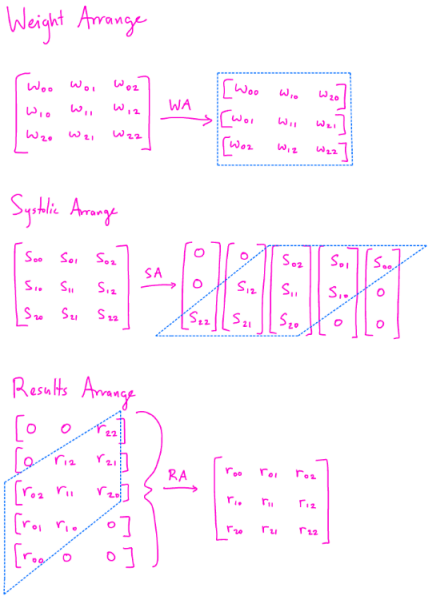

Figure 4. Breakdown of Arrange Block functionalities.

E. *Register File*

The TPU contains a centralized register file with a global bus interface to all of the main architectural blocks of the system. Unconventionally, the size of each register is $8N^2$ bits. This was done for simplicity in the control logic for data movement. We define K to be the number of registers in the register file, making the size of the register file $8N^2K$ bits.

F. *I/O Interface*

Our implementation uses the scan chain architecture for all I/O interfacing. The scan chain operates like a serial shift register (scan_reg) that can pause/unpause shifting. In our design, instructions are serially shifted into the scan_reg, and the host must assert the s_update signal to pause the shifting and begin the execution of the instruction (see Figure 5). The scan chain is very slow compared to other memory interfaces, such as PCIe. However, we had to choose this interface for

the ASIC implementation due to area and time constraints.

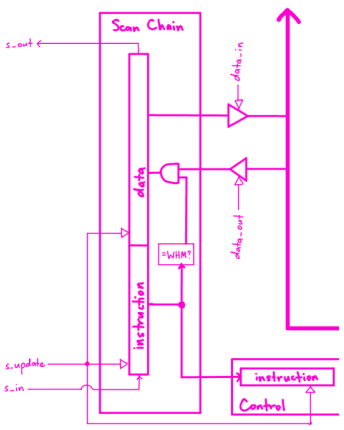

Figure 5. Scan chain microarchitecture.

G. Microarchitectural Bottlenecks

As the design was implemented, several microarchitectural limitations became apparent, restricting performance and efficiency at larger configurations. These bottlenecks highlight areas where further refinement is necessary to support more scalable and high-throughput implementations.

1. Quadratic Growth

Scaling the design with respect to N leads to quadratic growth in size relative to the systolic array. This non-linear scaling introduces significant overhead in terms of area and routing complexity, making the design less efficient and harder to manage as N increases.

2. Non-cascaded Systolic Inputs

A critical limitation arises from the lack of control logic for handling cascaded inputs into the Systolic Arrange block. Without dedicated mechanisms to coordinate these inputs, dataflow becomes less predictable and more difficult to optimize, reducing the potential performance benefits of systolic processing.

3. Scan Chain Memory Bound

Furthermore, the scan chain interface introduces a memory-bound bottleneck. Instead of being constrained by compute resources, the design is limited by memory throughput, preventing full utilization of the available computational power and undermining overall system performance.

IV. Design Implementation

A. FPGA Implementation

Artix-7 FPGA and measured using Vivado 2023 tools. The TPU design used N=4 and K=8. Table 2 summarizes the FPGA results.

FPGA PPA Metrics	
Critical Path	11.785 ns
Power	1.344 W
Area/Utilization	20800 LUTs, 41600 Registers

Table 2. Summary of FPGA TPU Implementation.

B. ASIC Implementation

Similarly, our RTL design was synthesized and implemented as an ASIC using TSMC 65nm technology nodes and Cadence/Synopsys design flows. Again, a TPU design of N=4 and K=8. Table 3 summarizes the ASIC results.

ASIC PPA Metrics	
Critical Path	1.915 ns
Power	12.86 mW
Area/Utilization	31254 um^2

Table 3. Summary of ASIC TPU Implementation.

C. Tradeoffs

Clearly, the ASIC outperforms the FPGA on all metrics. This is because custom technology at the transistor level eliminates the overhead of programmable logic. The ASIC offers a tailored datapath and control logic for maximum efficiency, unlike the FPGA, which relies on configurable interconnects and general-purpose resources. Moreover, the difference in power is quite substantial. However, to give some credit to the FPGA, post-RTL verification the FPGA implementation was instantaneous, while the ASIC implementation took approximately 2 months to develop.

V. TPU Pseudo-Compiler

To test a wide range of quantized neural network workloads on our custom TPU implementation, we developed a pseudocompiler. The core motivation was to bridge the gap between high-level machine learning models and our low-level TPU ISA by converting standard deep learning operations, specifically GEMM operations, into an executable instruction sequence that fits the TPU's ISA.

Our TPU assembler parses a set of instructions that specify data movement and matrix multiplication operations, using numpy array files as matrix sources for readability. Figure 5 provides an example of some assembly code that loads two matrices, multiplies them, and sends them back to the CPU.

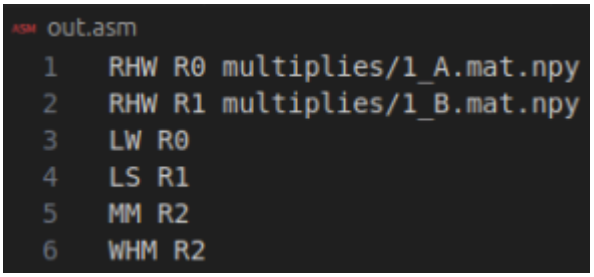

Figure 5. TPU Assembly Code.

The assembler operates under the assumption that all input matrices have been blocked to match the dimensions of the TPU's systolic array. Given the instruction sequence, it produces a corresponding binary that can be executed on the TPU for validation and benchmarking.

To evaluate the TPU on realistic workloads, we monkey-patched PyTorch's Conv2d and Linear layers. During inference, convolution operations were transformed into matrix multiplications using the im2col method, and all resulting GEMMs were logged. However, full model inference was not feasible due to two main limitations: the small systolic array size, and the lack of an addition instruction in our TPU's ISA. Since most neural networks rely on bias addition after each layer, the absence of such an instruction forced us to offload the intermediate results to the CPU for addition, and then reload them for the next multiplication. This round-trip introduced significant memory overhead, resulting in a severe memory bottleneck that prevented the TPU from delivering meaningful performance gains on end-to-end workloads.

VI. ML Workload Analysis

Given the architectural constraints discussed previously, we evaluated the performance of our TPU using theoretical analysis rather than full model execution. We selected three representative quantized models from PyTorch's pretrained suite: MobileNetV2, InceptionV3, and GoogLeNet. We analyzed them using our pseudocompiler. For each model, we computed the total number of required GEMM operations for the size of the TPU. We further estimated the execution time for inference for each model assuming that the CPU execution time was zero.

To estimate inference time, we accounted for the overhead introduced by our scan-chain memory interface, which imposes significant latency on data movement. We found that execution time for reading two matrices, multiplying them and returning the result to the CPU was given by the following equation:

$$Latency_{GEMM} = 24N^2 + 3N + 22$$

Where N is the size of the systolic array. The $24N^2 + 18$ cycles of latency correspond to two matrix reads and one write, while the remaining $3N + 4$ cycles cover the matrix multiplication and control logic overhead. Table 4 summarizes our findings by reporting the number of GEMM operations and the execution time in GCycles for each of the aforementioned models.

Model	Number of GEMMs	Estimated Runtime (GCycles)
MobileNetV2	1,958,656	9207.2
InceptionV3	128,000	393.2
GoogLeNet	64,000	98.3

Table 4. Workload Analysis

VII. Conclusion

In this work, we have presented a Google-inspired TPU microarchitecture accompanied by a psuedocompiler to explore the intricate aspects of hardware-software integration for machine learning workloads. Unlike existing high-level overviews, our approach delves into the microarchitectural details, addressing the design challenges associated with memory interfaces and system-level design. By implementing our design on both FPGA and ASIC platforms, we provide a comparative analysis of power, performance, and area (PPA) metrics, offering valuable insights into the trade-offs inherent in each implementation. While not aiming to exceed Google's TPU in performance, our contribution lies in demystifying the design space and enabling a deeper understanding of the architectural decisions critical to building efficient tensor computation hardware.

References

[1] N. P. Jouppi et al., "In-datacenter performance analysis of a tensor processing unit," 2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA), Toronto, ON, Canada, 2017, pp. 1-12, doi: 10.1145/3079856.3080246.

[2] J. Fowers et al., "A Configurable Cloud-Scale DNN Processor for Real-Time AI," 2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA), Los Angeles, CA, USA, 2018, pp. 1-14, doi: 10.1109/ISCA.2018.00012.

[3] Y. -H. Chen, T. Krishna, J. Emer and V. Sze, "14.5 Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks," 2016 IEEE International Solid-State Circuits Conference (ISSCC), San Francisco, CA, USA, 2016, pp. 262-263, doi: 10.1109/ISSCC.2016.7418007.

[4] J. Choquette, W. Gandhi, O. Giroux, N. Stam and R. Krashinsky, "NVIDIA A100 Tensor Core GPU: Performance and Innovation," in IEEE Micro, vol. 41, no. 2, pp. 29-35, 1 March-April 2021, doi: 10.1109/MM.2021.3061394.,