A TPU-like Design for FPGA Benchmarking

Aman Arora[*], Bagus Hanindhito[†], Harsh Gugale[‡], Lizy K. John[§]
Department of Electrical and Computer Engineering
The University of Texas at Austin
[*]aman.kbm@utexas.edu, [†]hanindhito@bagus.my.id, [‡]harsh.gugale@utexas.edu, [§]ljohn@ece.utexas.edu

***Abstract*—FPGA architectures have evolved over time from simple homogenous architectures comprising of mainly logic blocks (LBs) and on-chip memory (Block RAMs or BRAMs), to heterogenous architectures comprising of special purpose blocks like DSP slices, processing systems and DRAM controllers. With FPGAs being heavily used for accelerating AI and ML applications because of their configurability, improving FPGA architectures for ML/AI applications is required.**

Academic tools like VTR are used extensively by FPGA researchers around the world to explore FPGA architecture. Several modifications to FPGA architecture have been proposed in the recent years, but for careful analysis and exploration, we need benchmark designs that represent typical/common ML/AI workloads. Some such benchmarks have been created and used but are not yet publicly available.

Google TPU is a ASIC to accelerate AI/ML applications that has been deployed in the cloud as well the edge. We believe that adding a Google TPU-like design to the existing benchmark suite in VTR will be greatly helpful for researchers.

In this project, we develop a TPU-like design and take it through the VTR flow, with the intent of adding this design to the list of benchmarks in VTR. The design has a 8x8x8 matrix multiplier unit at its heart. We also prototype the matrix multiplier unit on a Xilinx FPGA board. We use the PYNQ framework to perform matrix multiplication on the FPGA.

***Index Terms*—TPU, FPGA, VTR, Benchmarking, Machine Learning, Hardware**

I. INTRODUCTION

Artificial intelligence and machine learning have become ubiquitous in today's world. Algorithms and models for these applications are getting more complex, and data sets are becoming larger and larger. As such, the computation needs are growing exponentially. Accelerating the computation required by AI/ML is a major challenge. Many solutions have been proposed and/or deployed for accelerating deep neural networks in hardware, ranging from ASICs (Google Tensor Processing Unit [8], DaDianNao [5]) to programmable GPUs ([13]) to configurable FPGA based solutions (Microsoft's Brainwave [6]). ASIC based designs have the best speed and power characteristics (fast and low power), but they lack configurability and adaptability which is crucial in the rapid changing world of AI/ML. GPU and CPU based designs, while highly programmable and adaptable, are not as fast and power-efficient as ASICs. FPGA based designs provide the best of both worlds. They provide massive parallelism, while being flexible and easily configurable, and also being fast and power-efficient.

A question naturally arises: Can we improve the performance of FPGAs for AI/ML? FPGA companies and researchers are exploring and deploying various techniques to make FPGAs better at accelerating AI/ML applications. These range from adding vector processors on the FPGA chip (Xilinx Versal [18]) to providing for integrating custom tensor tiles in the same package (Intel Agilex [7] [12]) to adding support for smaller ML-friendly precisions (like int4, fp16, etc.) in DSP slices on FPGAs. Boutros et al. [2] propose LB enhancements and add a shadow multiplier in LBs to increase MAC density in FPGAs improving deep learning performance. Boutros et al. [1] and Rasoulinezhad et al. [14] propose DSP slice modifications such as flexible precision and improvements to DSP-DSP interconnect.

FPGA devices mainly comprise of fine-grained programmable logic ("soft" LBs), embedded memory structures (BRAMs) and fixed-function math units ("hard" DSP slices). Coarser heterogeneous blocks like high speed IO controllers and processors are also seen on many FPGAs. Academic tools like VTR [11] are used extensively by FPGA researchers around the world to explore FPGA architecture. As mentioned above, several modifications to improve FPGA architectures for AI/ML have already been proposed in the recent years, but for careful analysis and exploration, we need benchmark designs that represent typical/common ML/AI workloads. VTR comes with a benchmark suite which includes designs from areas like image processing, packet processing, mathematics, etc. However, there are no benchmarks related to AI/ML, specifically neural networks. Some such benchmarks have been created and used [3] but are not yet publicly available.

In this project, we design a relevant AI/ML benchmark design, with the goal of contributing it to the VTR benchmark suite. We design a simpler version of the Google TPU [8]. We call it the TPU-like design. We believe that adding a TPU-like design to the existing benchmark suite in VTR will be greatly helpful to FPGA researchers for optimizing FPGA architecture for AI/ML applications.

Here is a summary of this project:

- We first develop a matrix multiplication unit (matmul), which is at the heart of the TPU design. We simulate and verify it.
- We deploy this matmul interfaced with other components like a DMA engine and BRAMs onto an FPGA to protoype this design.

- We develop and integrate other components in the TPU-like design like normalization unit, pooling unit and activation unit, and then simulate and verify it.
- We take this design through the VTR flow with a few FPGA architectures to ensure that our design works with VTR.

This report is organized as follows. Section II provides some background information of Google's TPU, Xilinx PYNQ and VTR. In Section III, we provide details of the hardware and software aspects of both our matrix multiplication design and the TPU-like design. Section IV describes the experimental methodology we used for this project. In Section V, we explain the results we've obtained. Finally, we conclude with closing statements and mention our future work in Section VI.

II. Background

A. Google's TPU

Google's TPU [8] (Tensor Processing Unit) is an ASIC for Machine learning inference/training in datacenters. The heart of TPU is a large systolic array matrix multiplier unit capable of delivering extremely high throughput when compared to CPU/GPU implementations.

1) Architecture: Figure 1, shows the high level architecture of the TPU. The TPU features a weight stationary architecture where separate large SRAM buffers feed data to the systolic array. It support operations such as accumulation, pooling and activation in hardware.

2) Programming Model: The TPU acts as an accelerator and a host CPU is responsible to schedule work and transfer data. TPU is programmed using a custom ISA. Instructions are transfered from the CPU through a PCIe interface.

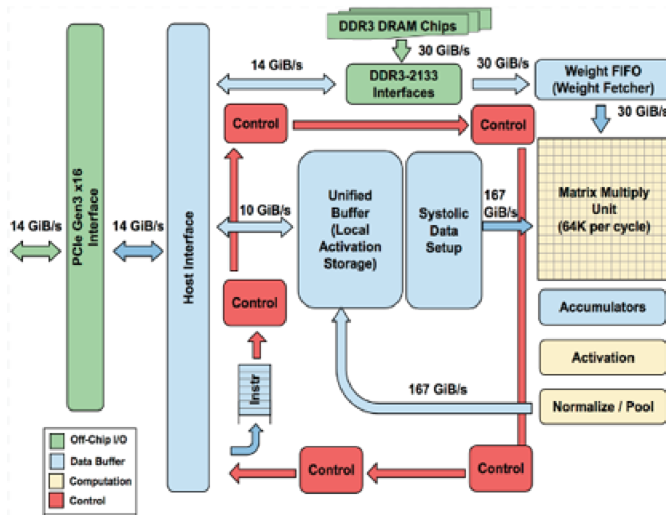

Fig. 1: Google TPU architecture [9]

B. VTR

VTR (Verilog To Routing) is an academic tool that enables exploration of FPGA architectures. VTR takes two inputs. The first input is an architecture description file, where the information of an FPGA's building blocks and interconnect resources is provided. The second input is a benchmark in form of a Verilog design that we intend to overlay onto the FPGA. VTR synthesizes, maps, places and routes the provided design on to the FPGA with the provided architecture. It uses tools called ODIN, ABC and VPR underneath for various parts of this process. The end result provided by VTR is area, timing, utilization reports that can be used by an FPGA architect to take informed decisions about the FPGA architecture. Figure 2 shows the steps followed by VTR.

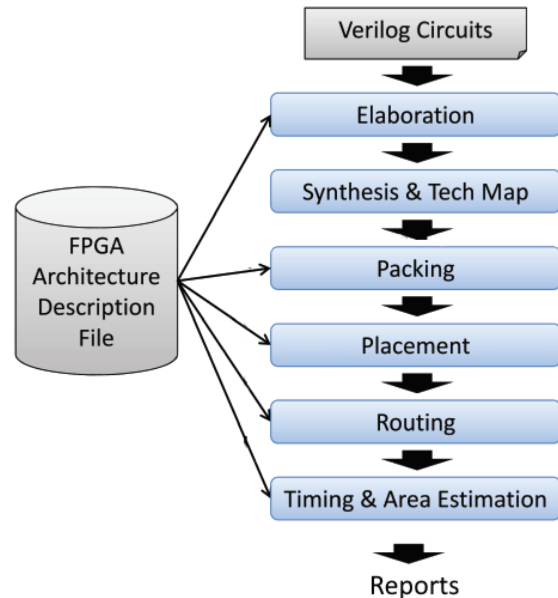

Fig. 2: VTR flow[11]

C. Xilinx PYNQ

PYNQ is an open-source project from Xilinx that makes it easier to use Xilinx FPGA platforms. Using the Python language and libraries, progammers can exploit the benefits of programmable logic and microprocessors to build more capable and exciting electronic systems. PYNQ exposes a Python interface to easily program and interact with the hardware overlayed on to the FPGA's programmable logic. It lowers the entry barrier for designer to use FPGAs and enable quick deployment and testing of designs. Figure 3 gives an overview of the PYNQ hardware and software stack. Some useful features include :

- Web server hosting the Jupyter Notebooks design environment.
- The iPython kernel and packages.
- Runs a stripped down version of Linux on the processor on the FPGA.
- Base hardware library support and API for the FPGA.

For prototyping the design in this project, we use PYNQ Z2 board from TUL [16].

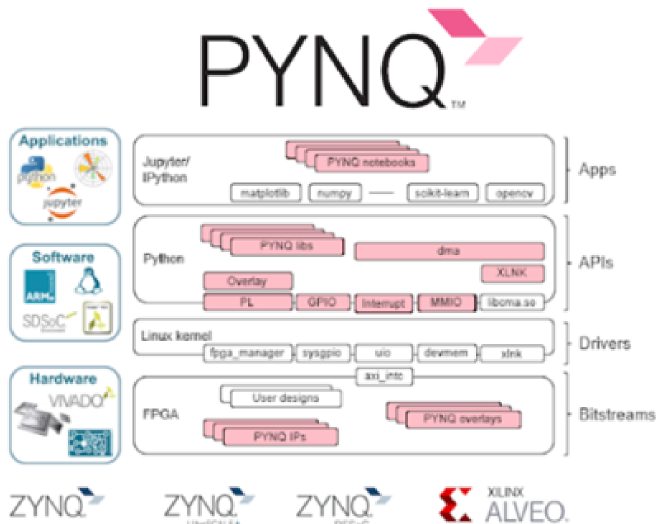

Fig. 3: Xilinx PYNQ software and hardware stack[17]

III. Architecture

A. Matrix multiplication on FPGA

We designed the core of the TPU-like design - the matrix multiplication unit - and prototyped it on an FPGA. Our design uses 8-bit integer format for storage as well as compute.

1) Hardware:

a) Top-level (Flow of information): Figure 4 shows a block diagram of the top-level of the design we overlayed onto the FPGA in the PYNQ board. A snapshot of this design from Xilinx Vivado can be seen in the Appendix (Figure 13). The matrices to be multiplied are allocated/stored on the DRAM. The Python program running on the Zynq Processing System (the ARM processor and its peripherals on the FPGA) triggers the DMA engine to transfer those matrices to the Dual-port BRAMs in the Programmable Logic. The Matrix Multiplication Unit reads these matrices from the BRAMs, multiplies them and stores the output back into a Dual-port BRAM. The Python program busy-waits on a status register in the Matrix Multiplication Unit until the operation has been completed. At that point, the DMA engine reads the output matrix from the BRAM and transfers the it to the DRAM. The busy-waiting could be improved to an interrupt based scheme, but that is left for future work.

We describe the details of each block in this diagram in the next few sections.

b) The matrix multiplication unit: The matrix multiplication unit we've designed uses a systolic [10] output stationary [4] architecture. We chose a systolic architecture because that's what is used on the TPU design, and also because systolic architectures reuse a piece of data multiple times without reading it again from memory, making them very efficient for compute-intensive tasks like matrix multiplication. We chose an output stationary design because Samajdar et al. [15] identify output stationary architectures to be better that weight stationary and input stationary.

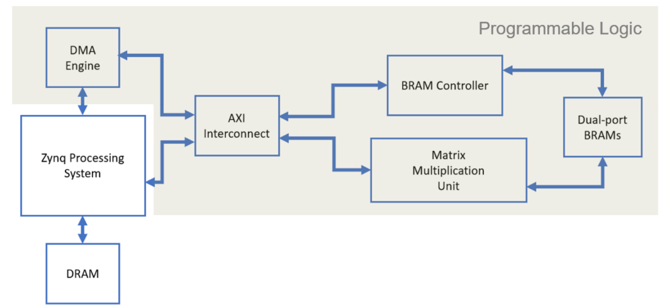

Fig. 4: Block diagram of the design overlayed onto the FPGA

The matrix multiplication unit (also called matmul), we prototyped on the FPGA, was a 4x4x4 unit (M=N=K=4). An MxNxK matrix multiplication unit multiplies an MxK matrix with a KxN matrix to generate a MxN matrix. There are 16 processing elements (PEs) in this design. Each PE basically performs a multiply-and-accumulate (MAC) operation.

Figure 5 shows some internal details of the matmul. The matmul's inputs and output are stored in BRAMs. Matrix A (input/activation matrix) is stored in the BRAM on the left. Matrix B (weight matrix) is stored in the BRAM on the top. The systolic data setup circuits fetch data elements from the BRAMs and stage them to enter the PEs in a systolic manner. The elements of Matrix B move from top to bottom and the elements of Matrix A move from left to right. The result *stays* in the respective PE until its computation is done. Once the results have been generated, the output interface logic shifts them out and writes them to the BRAM on the right.

One decision while implementing the matmul was regarding how to shift the outputs when they are available. Because of the systolic architecture of the design, the outputs get ready in a ”wave” that runs from the top-left to the bottom-right. One approach is to shift out outputs right when they become ready. This can save a few cycles, but then we pay the penalty to transpose/transform the data when writing to the memory so that it follows a column major order. Another approach is to wait until all outputs are available and then shift them. This appears to waste some cycles, but doesn't require any transposing/tranforming of the data. We chose the second approach.

In our implementation, input matrix A and C are stored in RAM in column-major order and input matrix B is stored in RAM in row-major order. For a 4x4x4 matmul, our implementation reads 8 input elements per clock cycle (4 elements of matrix A and 4 elements of matrix B). The grid of PEs contains 16 PEs.

c) The matmul IP: In order for the matmul to interact with the ARM processor on the FPGA on the PYNQ board, we needed to provide an AXI interface to it. To achieve this, we created an AXI slave IP with 10 registers using Xilinx Vivado IP generator and instantiated our matmul unit inside it. We created a state machine that interfaced the AXI registers to the matmul's inputs and outputs. These registers could be used

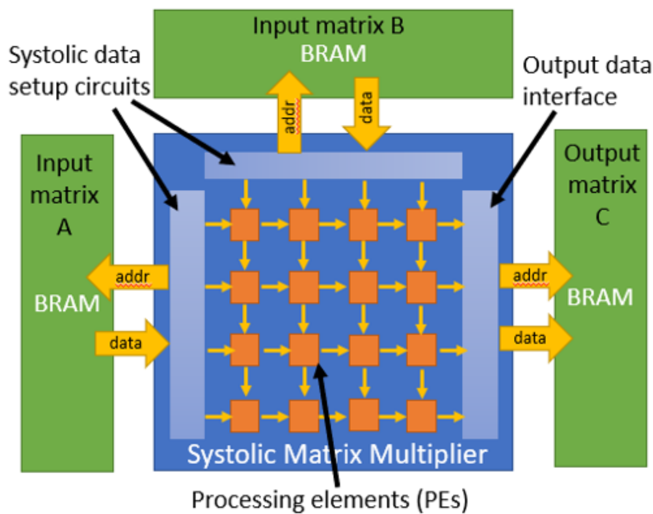

Fig. 5: 4x4x4 matrix multiplier design

to control the matmul and observe its state. We also exposed the BRAM interface to the top-most level of this block, so that we could interface a BRAM with it.

d) DMA engine: The DMA engine used in our design is called the AXI Central Direct Memory Access engine. It is an IP available from Vivado that can be easily instantiated and connected. This DMA engine has a memory mapped register interface that can be accessed from the ARM processor on the FPGA. We can configure the source address, the destination address and the size of the data to be transferred in these registers. To trigger the data transfer, a bit can be set in its register space. When the transfer is complete, a status bit is set and an interrupt is also generated.

There is another type of engine that is available in Xilinx's IP suite, called AXI Direct Memory Access. We tried using this IP first, but this IP supports a streaming interface, which wasn't a feature of our design. We wanted to transfer data between two memory mapped regions.

e) Other components: There were several other components required to built the complete system. To be able to transfer data to and from the BRAMs using a program running on the ARM processor or using a DMA engine, we needed to provide an AXI interface to the BRAMs. Xilinx Vivado provides an AXI BRAM Controller IP that can be instantiated and connected to the BRAMs. An AXI switch was required to connect multiple masters in the system (the ARM processor and the DMA engine) to the multiple slaves in the system (the BRAMs and the matmul). A reset generation block was also instantiated to provide resets to the various blocks in the design.

2) Software: The beauty of the PYNQ [17] paradigm is that it allows a user to control and observe the hardware overlayed onto the FPGA using Python. Traditionally, the interface between the hardware overlayed onto the FPGA and software running on the processor on the FPGA is challenging. But PYNQ makes it easy. A library of Python code provides easy APIs and classes to access the hardware overlayed on the FPGA. The hardware components in the design become available to the programmer as Python objects, which are very easy to manipulate and observe.

a) Drivers: Although writing a driver is not necessary to be able to interface with the hardware overlayed on the FPGA, it provides a meaningful level of abstraction and allows for code reuse. We created 3 drivers - one for the BRAM, one for the DMA engine and one for the Matmul unit. These drivers provide convenient APIs to perform operations like triggering the DMA, writing data to an address in the BRAM, reading the status of the matmul unit, etc. To see the code of the drivers, the reader is referred to the Appendix (Listing 6, Listing 5 and Listing 7)

b) do_matmul routine: We coded a routine that encapsulates the sequence of operations required to perform the matrix multiplication on the FPGA. It has 3 arguments - first two are the matrices to be multiplied and the third one contains the result after the routine has finished. The operations that this routine performs are similar to the sequence of steps described in Section III-A1a. Listing 1 shows the code of this routine.

Listing 1: Code for do_matmul

```
1  def do_matmul(a,b,c):
2      matmul.reset()
3      dma.reset()
4      dma.do_transfer(a.device_address, bram_a.mmio.
       base_addr, a.nbytes) #sent to bram_a
5      dma.do_transfer(b.device_address, bram_b.mmio.
       base_addr, b.nbytes) #sent to bram_b
6      matmul.start()
7      while not matmul.is_done():
8          pass
9      matmul.clear_done()
10     dma.do_transfer(bram_c.mmio.base_addr, c.
       device_address, c.nbytes) #bring from bram_b
```

c) Top-level code: The top-level code that uses the *do_matmul*() to perform the matrix multiplication is shown in Listing 2.

Listing 2: Top level code for the matrix multiplication

```
1 a = allocate(shape=(4,4), dtype=np.uint8)
2 b = allocate(shape=(4,4), dtype=np.uint8)
3 c = allocate(shape=(4,4), dtype=np.uint8)
4 for i in range(4):
5     for j in range(4):
6         a[i,j] = random.randint(0,9)
7         b[i,j] = random.randint(0,9)
8 do_matmul(a,b,c)
9 print(a,b,c)
```

B. TPU-like design

Figure 6 shows our TPU-like architecture. A matrix multiplication unit is at the heart of the TPU-like design, just like the TPU design. We used the matmul block described above, extended it to a 8x8x8 matmul and then used it in our TPU-like design. The choice of an 8x8x8 matmul is arbitrary and serves as a proof of concept that our matmul design is flexible and easily scalable. Our design uses 8-bit integer format for storage as well as compute. We only support inference.

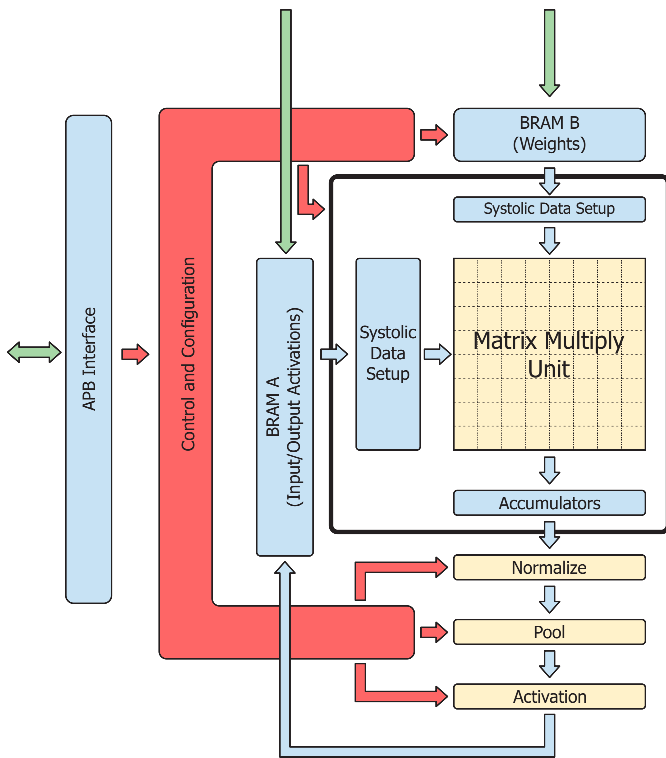

Fig. 6: TPU-like architecture

1) Hardware: The hardware architecture of the TPU-like design is similar to the Google TPU architecture (Figure 1), with a few key differences:

- In the TPU architecture, a PCIe interface is used to send commands to the accelerator. In the TPU-like architecture, the commands are send through an AMBA APB interface.
- Instead of data being read from a DDR3 interface like in the TPU architecture, our TPU-like architecture uses raw BRAM interfaces. These interfaces, shown as vertical green arrows, are connected to the testbench.
- The unified buffer and weight FIFO in the TPU architecture are implemented using SRAMs. In our architecture, these are replaces by BRAMs.

a) Top-level (Flow of information): Here we explain the datapath and the control path of our TPU-like at the top-level:

- Input data is loaded into BRAM A and weights are loaded into BRAM B. This is done by ports of the BRAMs exposed to the testbench. In a real system, these ports would connect to a DRAM.
- Commands are sent through an AMBA APB interface, in form of register writes. These commands configure the various blocks of the TPU and also trigger the TPU's operation.
- The systolic data setup logic blocks read the elements stored in the two BRAMs and stage them into the matmul unit.
- When matmul finishes its operation, the output is either stored in accumulators (see Section III-B1f for details on when this is required) or shifted out to the next unit.
- Normalization, pooling and activation operations are done on the output shifted out of the matmul in that order. Each of these units can be disabled by sending appropriate commands.
- The final result is stored back in BRAM A and the process can be repeated as desired.

b) Configuration block: The Configuration block in the design holds multiple control and status registers. This block has an AMBA APB interface. We chose an APB interface because of its simplicity. APB interface is a slow interface, but that is okay because we are not transfering any data using this interface. We are only transfering commands over this interface. Writing to a control register is similar to sending a command to the accelerator. There are many registers in this block, for example, a register to configure the base address of weights matrix, a register to enable/disable various blocks in the design, etc.

c) Control logic: The Control logic is basically a Finite State Machine (FSM). This block orchestrates the operation of the whole accelerator. This block triggers each block in the design when its inputs are ready. This block receives various signals from the Configuration block which control the behavior of the FSM.

d) Matrix multiplication unit: The matrix multiplication unit's size is 8x8x8. That is, it supports multiplying an 8x8 matrix with another 8x8 matrix to produce an 8x8 matrix. We used the matrix multiplication unit described in Section III-A for our TPU-like design, with one difference. Instead of the output being written to a third BRAM, the output is just passed downstream to the normalization, pooling, activation blocks and then written back to BRAM A. This is done for two reasons: (1) Power and time is saved because these downstreams blocks don't have to re-read the outputs written into that BRAM, and (2) Output of one layer becomes the input of a subsequent layer which is to be read from BRAM A.

e) Systolic data setup: There are two systolic data setup units in our design - one for each BRAM. These block generate addresses of which elements are to be fetched from the BRAM. We read N elements from each BRAM at a time, where N is the size of the matmul. However, not all the N elements have to be sent to the matmul unit. So, various elements are delayed with different amounts. For example, let's consider the elements read from BRAM A that are sent into the matmul unit from left to right. The element entering the first row is sent right away, the element entering the second row is delayed by 1 clock, the element entering the third row is delayed by 2 clocks and so on. This staging follows from the systolic operation of the design.

f) Accumulators: Although the martix multiplication unit in the TPU-like design is small (8x8x8), it can handle multiplication of larger matrices that are common in neural networks. To make this possible, storing partial sums is required so that they can be accumulated to obtain the final result. This is

done in the Accumulators block. This unit contains 64 8-bit accumulators.

Figure 7 shows how the accumulation is done. The accumulator unit can operate in 3 modes - Disabled, Save and Add. In the Disabled mode, this unit is just bypassed. In the Save mode, the current results of the matrix multiplication are saved in registers. In the Add mode, the current results of the matrix multiplication are added with the existing (previous) results in the registers.

One of the decision points while designing the accumulator block was regarding how much hardware should we provide to handle larger matrices. A purely software approach would mean we only support multiplying the matrices of the size of the matmul and all accumulation is done in software. A purely hardware approach would mean tiling the output matrix in hardware, doing the accumulation and also making the hardware aware of the size of matrices being multiplied. Our approach is somewhere in the middle of these two approaches.

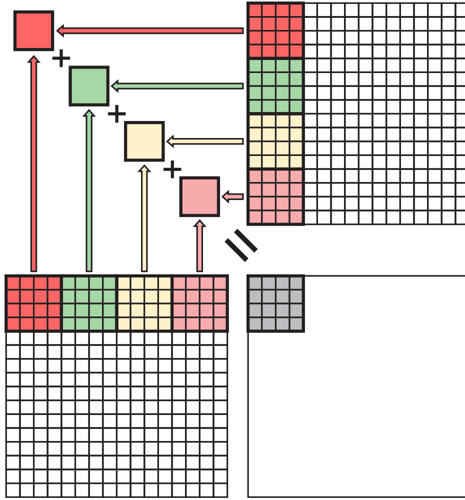

Fig. 7: Accumulators are needed when matrices to be multiplied are larger that the matmul in the design

g) Normalization unit: This unit is used to normalize the inputs coming in, so that the resulting data (along the Batch dimension) has a mean of 0 and variance of 1. For doing this, we need to subtract the mean from each value in the input and divide it with the variance.

The process of normalization for Training and Inference are very different. For training, the mean and variance values are to be calculated on the current data (we call this the ”stats” step) and then applied to the data to normalize it as well (we call this the ”apply” step). However, for inference, the mean and variance values from training are used and only the ”apply” step is required. In our design, we only support inference and hence we only perform the ”apply” step. The mean and variance are provided as inputs and can be configured in the registers in the Configuration block.

Normalization requires one addition and one division for each element in the input. Division is a costly operation to implement in hardware. Division can be avoided if we only support inference. This is because division by variance is equivalent to multiplying by the inverse of variance. So, if we supply the value for inverse of variance as input, instead of the value of variance, we can just perform an addition and a multiplication and reduce the complexity of this block. We use this optimization in our design.

h) Pooling unit: Pooling is often used in Neural networks, to reduce the dimensionality of the input layer being fed to the next layer. We implement software controlled pooling operation in our TPU-like design. The input to the Pooling block is fed from the output of the normalization block. Some features of this block are:

- Pooling operations is done in batches. As one output batch is normalized, it is fed for pooling operation. If pooling is not enabled, the input is passed to the output without modification.
- We implement average pooling in our design and give the user 3 separate pooling windows to choose from - 1, 2 and 4. This implies, pooling block can reduce the dimensionality of the output by upto 4x.
- Output bus data width is fixed, thus if windows of 2 or 4 are used, the output is padded with zeros.

i) Activation unit: In our TPU design, we implement two activation functions that are often used in real workloads: the rectified linear unit (ReLU) and the hyperbolic tangent (TanH). The activation unit can be configured to use one of these activation function by setting the activation unit control and status register. Both of the activation functions that we support are briefly explained below.

- The Rectified Linear Unit (ReLU)
  This is one of the most widely used activation functions, especially in convolutional neural networks. ReLU will output zero if the input is less than zero, otherwise ReLU will forward the positive input as its output. The hardware implementation of ReLU is simple as it only consists of one multiplexer and one comparator.
  The advantages of having ReLU as activation function is that it is cheap to implement in hardware since it is based on simple mathematical function. ReLU tends to converge faster because there is no input saturation problem as input gets large and there is no vanishing gradient problem. Lastly, it is sparsely activated since it will not be activated for all of the negative inputs.

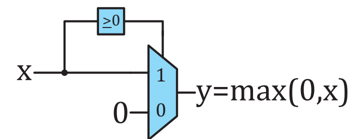

Fig. 8: Hardware Implementation of ReLU

- The Hyperbolic Tanget (TanH)
  This activation function is more complex to implement compared to the ReLU. The TanH function is based on the hyperbolic tangent function and solves the ”stuck” problem of the logistic sigmoid function. While the

output of logistic sigmoid function is between 0 to 1, the TanH has output between -1 to 1 and thus strongly negative inputs will be mapped to the negative outputs. This property reduce the likelihood of the network to get stuck during training. The TanH function is mainly used for classification between two classes and used in feed-forward networks.

There are two problems that we have to address before we implement the TanH activation function into our design. First, the output range of the TanH that only has range between -1 to 1. Since our TPU design uses int8 data format, direct adoption of TanH will give the output of three possible values: -1, 0, and 1. To solve this problem, we map the output values in the range of -1 to 1 into the range of -128 to 127 by multiplying the output with 128.

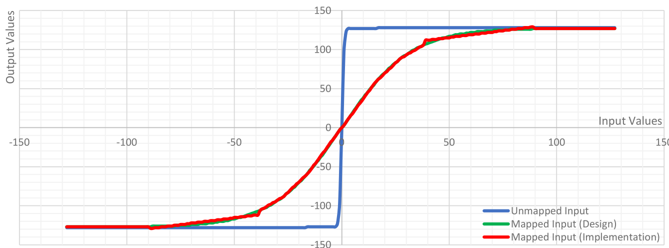

Fig. 9: Input Map, Output Map, and Function Approximation of TanH

Second problem is the dynamic range of the input, which is between -128 to 127. Direct adoption of TanH will produce 25 different output values since the input below -13 will have output of -1 and the input above +13 will have output of 1. To avoid this problem, we map the input values in the range of -128 to 127 into the range of -4 to 3.96875 by dividing the input with 32.

TABLE I: Piece-Wise Linear Approximation of TanH Activation Function

Input Range	Slope	Y-Intercept	Linear Function
$x \geq 90$	0	127	$y = 127$
$39 \leq x < 90$	0	99	$y = 99$
$28 \leq x < 39$	2	46	$y = 2x + 46$
$16 \leq x < 28$	3	18	$y = 3x + 18$
$1 \leq x < 16$	4	0	$y = 4x$
$x = 0$	0	0	$y = 0$
$-16 < x \leq -1$	4	0	$y = 4x$
$-28 < x \leq -16$	3	-18	$y = 3x - 18$
$-39 < x \leq -28$	2	-46	$y = 2x - 46$
$-90 < x \leq -39$	0	-99	$y = -99$
$x \leq -90$	0	-127	$y = -127$

After we have agreed on how the input and the output of the TanH function are mapped, the next step is to implement the TanH function in hardware. We use piece-wise linear approximation of the TanH function using a linear function in the form of ax+b. We divide the input range of the TanH function into 11 segments, each of them has its own linear function. The hardware implementation will consist of single multiplier, single adder, two 11-entry LUTs, and two LUT's address decoders. The LUT address decoder consists of range comparator which determine the appropriate LUT entry for a given input. We also added buffers between the multiplier and adder to shorten the critical path of the circuit.

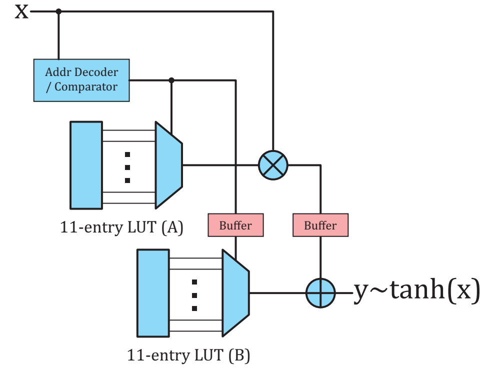

Fig. 10: Hardware Implementation of TanH

2) Software: Software is required to execute a neural network on the TPU-like design. Since, we only simulate this design, the software for this case takes the form of the test sequences that we wrote. We developed a simple testbench with two tests in it (each test can have many variations though).

a) Layer Test: The layer test runs a two-layered fully-connected network through our TPU-like design. In this test, the sizes of layers are such that they match the size of the matmul. Specifically, for this test for both layers, number of inputs = 8, batch size = 8, number of neurons = 8 and number of outputs = 8. Listing 3 shows the pseudocode of this test.

Listing 3: Pseudo code of the layer test

```
1: Initialize BRAMs A and B with the input matrix
    and the weight matrix respectively
2: Main loop
  2.1: Enable required blocks (normalization, pool,
    activation) by configuring the ENABLES register
    2.1.1: Configure the mean and variance values
    for the Normalization block
    2.1.2: Configure the pooling window size for the
     Pooling block
    2.1.3: Configure the activation type (relu or
    tanh) for the Activation block
  2.2: Configure the start addresses of matrices A,
    B and C in the BRAMs
  2.3: Trigger the TPU by writing to the START
    register bit
  2.4: Wait until the TPU is done by polling the
    DONE register bit
  2.5: Repeat steps 2.2, 2.3, 2.4 for next layer
3: Compare the outputs with the expected golden
    output
```

b) Accumulator Test: The accumulator test runs a two-layered fully-connected network through our TPU-like design. In this test, the sizes of the layers are larger than the matmul size. Specifically, for this test for both layers, number of inputs = 24, batch size = 24, number of neurons = 24 and number of outputs = 24. The matmul size is 8x8x8. Listing 4 shows the pseudocode of this test.

Listing 4: Pseudo code of the accumulator test

```
1: Initialize BRAMs A and B with the input matrix
    and the weight matrix respectively
2: Main loop
  2.1: Enable required blocks (normalization, pool,
    activation) by configuring the ENABLES register
    2.1.1: Configure the mean and variance values
    for the Normalization block
    2.1.2: Configure the pooling window size for the
     Pooling block
    2.1.3: Configure the activation type (relu or
    tanh) for the Activation block
  2.2: Configure the address strides in the STRIDES
    registers. For a 24x24 input, this value is 24.
  # Tile size is the same as matmul size. So, for as
     24x24 input with an 8x8 matmul, each tile will
    need 3 matrix multiplication passes to be done.
  2.3 For each output tile:
    2.3.1: For each pass:
      2.3.1.1: Configure the start addresses of
    matrices A, B and C in the BRAMs
      2.3.1.2: Trigger the TPU by writing to the
    START register bit
      2.3.1.3: Wait until the TPU is done by polling
     the DONE register bit
  2.4 Repeat steps 2.2, 2.3, 2.4 for next layer
3: Compare the outputs with the expected golden
    output
```

IV. Experimental methodology

A. Matrix multiplication on FPGA

We implemented the matrix multiplication unit in Verilog at RTL-level. At first, we experimented with small-sized 4x4 matrix multiplication as an effort to familiarize our self with the design flow of the PYNQ board. Our matrix multiplication unit accepts 8-bit signed integer data format and has basic structure of a systolic array. Before we implemented the design on the FPGA, we verified its functionality using Synopsys VCS/DVE.

We used Xilinx Vivado for implementing the matrix multiplication unit on the PYNQ Board. We used some IPs provided by Xilinx to implement our design, including the DSP blocks for multiplication and accumulation and the DMA block for transferring data between CPU and FPGA. The PYNQ Board that we use is the PYNQ Z2 board with Xilinx Zynq Z7020 SOC. The SOC consists of dual-core ARM Cortex-A9 processors and Artix-7 Programmable Logic (FPGA) with 512MB DDR3 memory. The Artix-7 FPGA features 85,000 programmable logic cells, 53,200 look-up tables, 106,400 Flip-Flops, 4.9 MbBlock RAM in 140 36kB-blocks, and 220 DSP Slices. We run Ubuntu 18.04 on the ARM cores and has the Jupyter Notebook server activated by default. Then, we create the driver (firmware) for our matrix multiplication unit in Python. Finally, we compared the performance of our matrix multiplication unit with the performance of the ARM cores to multiply the same size of matrices.

B. TPU-like design for VTR

We implemented our TPU-like design in Verilog at RTL-Level. Our TPU-like design includes the matrix multiplication unit that we had implemented in PYNQ board before. But, in VTR, we wanted to try to use larger matrix multiplication. Instead of manually writing more codes to implement larger matrix multiplication unit, we created a Python-based script to generate larger matrix multiplication units. This script will automatically generates the Verilog files based on the size of the matrix multiplication units that is given. Then, as usual, we verify the functionality of our TPU-like design using Synopsys VCS/DVE.

We took our TPU-like design through the VTR flow. We used four different FPGA architectures in our experiments. These FPGA architectures are listed below.

- k6_frac_N10_mem32K_40nm
  We refer to this architecture as Architecture A. This architecture has similar design as the Intel Stratix IV 40nm FPGA. It has fracturable 6-LUT based LBs, with 32K BRAM blocks and 36x36 multipliers in DSP slices.
- k6_frac_N10_frac_chain_mem32K_htree0_routedCLK-_40nm
  We refer to this architecture as Architecture B. In this architecture, the LBs have carry chain links to vertically adjacent LBs. This architecture also has clock routing information in it as well.
- k6_frac_N10_4add_2chains_tie_off_depop50_mem20K-_22nm
  This is a 22 nm architecture and has two separate carry chains in a LB with separate and end points. This architecture has 27x27 multipliers in the DSP blocks and also 20K BRAM blocks. The crossbars on the LBs are 50% depopulated. We refer to this architecture as Architecture C.
- k6_N8_ripple_chain_gate_boost_0.2V_22nm
  We refer to this architecture as Architecture D. This architecture has 8 6-LUTs in 1 LB and uses gate-boosting in the pass-transistors in its switch boxes and connection boxes.

Then, we compared the resources utilization and the timing of our TPU-like design implementation on different architectures of FPGA.

V. Results

A. Matrix multiplication on FPGA

1) FPGA implementation results: We have successfully implemented our matrix multiplication unit in PYNQ Board FPGA. We also developed a driver (firmware) in Python to easily operate our matrix multiplication unit. At first, the Xilinx Vivado did not map our multiplication and accumulation into DSP slices in the FPGA. By default, it maps multipliers and adders to LBs. To use the DSP slices, we need to define a design constraint in XDC file which will guide Vivado to map the multiplication and addition into DSP slices.

As we can see from the resource utilization table, our 4x4 matrix multiplication unit uses 16 DSP slices. The number of DSP slices that are being used is associated with the number of processing elements that our matrix multiplication unit has. The 4x4 matrix multiplication unit will have 4x4=16 processing elements, and hence 16 DSP slices.

TABLE II: Resource Usage of Synthesized Design in PYNQ Board

Resource	Utilization	Available	% Utilization
LUT	8526	53200	16.03
LUTRAM	620	17400	3.56
FF	9397	106400	8.84
BRAM	6	140	4.29
DSP	16	220	7.27

Furthermore, Vivado was able to synthesize and implement our design by fulfilling all of the timing requirements. After implementation finished, there is no setup and hold violation. Our design is able to run at target speed of 100MHz.

TABLE III: Timing Report of Implemented Design in PYNQ Board

Parameter	Setup	Hold
Worst Slack (WS)	1.152ns	0.052ns
Total Slack (TS)	0.000ns	0.000ns
Number of Failing Endpoints	0	0
Total Number of Endpoints	8097	8097

2) Acceleration obtained using the FPGA: To connect our matrix multiplication unit to the embedded ARM core, we implemented two different approaches: using memory-mapped IO (MMIO) and using CDMA. The MMIO approach is slower than the CDMA approach for large matrices size. Finally, we also compare the performance of our matrix multiplication unit with the CPU-implementation of matrix multiplication when multiplying 4x4 matrices. Our matrix multiplication unit finished the work in 0.0047s while the ARM core finished the work in 0.0011s. This means that the CPU-implementation is faster than our matrix multiplication unit. The reason behind this is that the size of the matrices is too small to amortize the time needed to transfer the data between CPU and FPGA. With larger matrices size and CDMA interface, our matrix multiplication unit should perform faster than the CPU-implementation.

```
print("Result from overlay running on fpga:")
start_time = time.time()
do_matmul(a,b,c)
end_time = time.time()
print(c)
print("Time taken = ", (end_time-start_time))

Result from overlay running on fpga:
[[30 20 32 30]
 [ 9  6 12 12]
 [31 20 25 20]
 [36 23 36 26]]
Time taken =  0.0046885013580322266

print("Result from numpy running on cpu:")
start_time = time.time()
c_cpu = np.matmul(np.transpose(a),b
end_time = time.time()
print("c=", c_cpu)
print("Time taken = ", (end_time-start_time))

Result from numpy running on cpu:
c= [[30 20 32 30]
 [ 9  6 12 12]
 [31 20 25 20]
 [36 23 36 26]]
Time taken =  0.001104116439819336
```

Fig. 11: CPU and FPGA Matrix Multiplication Performance Comparison

B. TPU-like design

1) Speeds and Feeds: Table IV compares the speeds and feeds of the Google TPU design with our TPU-like design. We can see that our design is significantly smaller and has less performance than the Google TPU. However, we didn't intend the two designs to be similar on these aspects. The only thing these designs share is the similar architecture.

The peak math throughput comes from our design's ability to perform 64 int8 MAC operations at 150 MHz. See V-B2 for clock frequency that our design can operate on. The on-chip memory size comes from our design having two 64-bit wide, 2048 deep BRAMs.

In the table, we mention the DRAM bandwidth of our TPU-like design to be "NA" (Not Applicable). That's because we don't have a DRAM interface in our design. We connect the BRAMs to the testbench. Data is loaded into the BRAMs before operation is started. We don't model/analyze the time taken for DRAM loads and stores.

TABLE IV: Speeds and feeds of the Google TPU design and our TPU-like design

Parameter	Google TPU	Our TPU-like design
Technology node	28 nm	40 nm
Peak math throughput	92 TFlops	19.2 GFlops
Clock frequency	700 MHz	150 MHz
DRAM bandwidth	34 GB/s	NA
On-chip memory	28 MB	32 KB

2) VTR results for various FPGA architectures: Table V shows the timing, resource usage and area results obtained from VTR for various FPGA architectures. The critical path delay for our design ranges from about 6.6ns to 7.6ns. That means the clock frequency of our design can range from 151 MHz to 131 MHz. Looking at the reports, we saw that the critical path contained a multiplier and some combinatorial logic in the activation block. That was expected. But we are not sure about why the delays are similar in the 22nm architectures and the 40nm architectures. A quick look at the architectures makes us think that the 22nm architectures don't have scaled-down delays in them (for eg. delay of a multiplier is the same in both 40nm and 22nm architectures). This needs to be investigated further.

We can see that the number of DSP slices used in all architectures is 74. These architectures don't have adders in their DSP slices, only multipliers. We were expecting a total of 64 (8*8 in the matmul) + 8 (in normalization block) + 8 (in activation block). But looking carefully at the final netlist generated by VTR, we saw that 74 came from 64 (8*8 in the matmul) + 2 (4 normalization multipliers were mapped to 1 DSP block) + 8 (in activation block).

Regarding BRAMs, we only instantiated 2 BRAMs in our design - one for matrix A and one for matrix B. But we saw 8 BRAMs used in the report. We analyzed why that was the case. In our design, we used a BRAM with 64-bit datawidth and 2048 depth (11 address bits). In the FPGA architectures, however, multiple BRAM configurations were available like 32 wide + 1024 deep, 16 wide + 2048 deep, 8 wide +

4096 deep, etc. VTR automatically combined 4 BRAMs in the configuration "16 wide and 2048 deep" to create a wider BRAM for us. That's why there are 8 BRAMs in the results and not 2.

TABLE V: Timing and Resource Usage of Implemented Design in VTR (MWTA stands for Minimum Width Transistor Area)

Parameter	Architecture			
	A	B	C	D
VTR elapsed time (s)	123.26	152.04	192.58	140.48
Number of LBs	519	500	749	997
Number of Memories	8	8	8	8
Number of DSP slices	74	74	74	74
Number of Primary Inputs	213	213	213	213
Number of Primary Outputs	161	161	161	161
Minimum Channel Width	70	74	64	68
Logic Block Area (MWTA)	6.17e+07	6.06e+07	4.43e+07	4.62e+07
Critical Path Delay (ns)	7.6222	7.4183	7.3017	6.65653
Routing Area (MWTA)	1.78e+07	1.85e+07	2.26e+07	1.53e+07

VI. Conclusion

During this project, we developed a systolic matrix multiplication unit and deployed it on the PYNQ Z1 board. We wrote software in Python to interact with this unit. We then added onto to this design to create a Google TPU-like design. We took this design through VTR flow for various FPGA architectures. We plan to integrate this design as a AI/ML benchmark into the next version of VTR so that it can be used by the larger academic community for FPGA research.

Here's the future work we've identified before we can submit this design to be used as a benchmark to VTR:

- We need to perform more exhaustive testing of our design. So far or inputs have only had positive values. We need to add negative values to inputs and weights. The input and the weight matrices so far have been square. We want to write tests that use non-square matrices.
- We've only used this design with fully-connected networks. We plan to run a convolutional neural networks through this design.
- For now we only support one precision - int8. We plan to change this to 8-bit fixed point. Also, we have though of implementing mixed precision, where the compute is done in a higher precision compared to the storage.

References

[1] A. Boutros, S. Yazdanshenas, and V. Betz, "Embracing diversity: Enhanced dsp blocks for low-precision deep learning on fpgas," in *2018 28th International Conference on Field Programmable Logic and Applications (FPL)*, 2018, pp. 35–357.

[2] A. Boutros, M. Eldafrawy, S. Yazdanshenas, and V. Betz, "Math Doesn't Have to be Hard: Logic Block Architectures to Enhance Low-Precision Multiply-Accumulate on FPGAs," 02 2019, pp. 94–103.

[3] A. Boutros, S. Yazdanshenas, and V. Betz, "You cannot improve what you do not measure: Fpga vs. asic efficiency gaps for convolutional neural network inference," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 11, no. 3, Dec. 2018. [Online]. Available: https://doi.org/10.1145/3242898

[4] Y. Chen, J. Emer, and V. Sze, "Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks," in *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, 2016, pp. 367–379.

[5] Y. Chen *et al.*, "DaDianNao: A Machine-Learning Supercomputer," in *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-47. Washington, DC, USA: IEEE Computer Society, 2014, pp. 609–622. [Online]. Available: http://dx.doi.org/10.1109/MICRO.2014.58

[6] J. Fowers *et al.*, "A Configurable Cloud-scale DNN Processor for Real-time AI," in *Proceedings of the 45th Annual International Symposium on Computer Architecture*, ser. ISCA '18. Piscataway, NJ, USA: IEEE Press, 2018, pp. 1–14. [Online]. Available: https://doi.org/10.1109/ISCA.2018.00012

[7] Intel. (2019) Intel Agilex FPGAs and SOCs. [Online]. Available: https://www.intel.com/content/www/us/en/products/programmable/fpga/agilex.html

[8] N. P. Jouppi *et al.*, "In-Datacenter Performance Analysis of a Tensor Processing Unit," *CoRR*, vol. abs/1704.04760, 2017. [Online]. Available: http://arxiv.org/abs/1704.04760

[9] ——, "In-Datacenter Performance Analysis of a Tensor Processing Unit," *CoRR*, vol. abs/1704.04760, 2017. [Online]. Available: http://arxiv.org/abs/1704.04760

[10] H. T. Kung, "Why Systolic Architectures?" *Computer*, vol. 15, no. 1, pp. 37–46, Jan. 1982. [Online]. Available: https://doi.org/10.1109/MC.1982.1653825

[11] J. Luu *et al.*, "VTR 7.0: Next Generation Architecture and CAD System for FPGAs," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 7, no. 2, pp. 6:1–6:30, June 2014.

[12] E. Nurvitadhi, S. Shumarayev, A. Dasu, J. Cook, A. Mishra, D. Marr, K. Nealis, P. Colangelo, A. Ling, D. Capalija, and U. Aydonat, "In-Package Domain-Specific ASICs for Intel® Stratix® 10 FPGAs: A Case Study of Accelerating Deep Learning Using TensorTile ASIC," 02 2018, pp. 287–287.

[13] NVIDIA. (2017) NVIDIA TESLA V100 GPU ARCHITECTURE. [Online]. Available: https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf

[14] S. Rasoulinezhad, H. Zhou, L. Wang, and P. H. W. Leong, "Pirdsp: An fpga dsp block architecture for multi-precision deep neural networks," in *2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2019, pp. 35–44.

[15] A. Samajdar, Y. Zhu, P. N. Whatmough, M. Mattina, and T. Krishna, "Scale-sim: Systolic CNN accelerator," *CoRR*, vol. abs/1811.02883, 2018. [Online]. Available: http://arxiv.org/abs/1811.02883

[16] TUL. [Online]. Available: http://www.tul.com.tw/ProductsPYNQ-Z2.html

[17] Xilinx. (2018) Pynq. [Online]. Available: https://github.com/Xilinx/Pynq

[18] ——. (2018) Xilinx AI Engines and Their Applications. [Online]. Available: https://www.xilinx.com/support/documentation/white_papers/wp506-ai-engine.pdf

Appendix

All the code for this project is located on Github: *https://github.com/aman26kbm/tpu_like_design* If you need permissions to view the code, please email the authors.

Listing 5: Code for the BRAM driver

```
class BramDriver(DefaultIP):
    def __init__(self, description):
        super().__init__(description=description)

    bindto = ['xilinx.com:ip:axi_bram_ctrl:4.1']

    def write_a(self, a):
        bram_a.write(0,  int((a[3,0]<<24) + (a[2,0]<<16) + (a[1,0]<<8) + (a[0,0])))
        bram_a.write(4,  int((a[3,1]<<24) + (a[2,1]<<16) + (a[1,1]<<8) + (a[0,1])))
        bram_a.write(8,  int((a[3,2]<<24) + (a[2,2]<<16) + (a[1,2]<<8) + (a[0,2])))
        bram_a.write(12, int((a[3,3]<<24) + (a[2,3]<<16) + (a[1,3]<<8) + (a[0,3])))

    def write_b(self, b):
        bram_b.write(0,  int((b[0,3]<<24) + (b[0,2]<<16) + (b[0,1]<<8) + (b[0,0])))
        bram_b.write(4,  int((b[1,3]<<24) + (b[1,2]<<16) + (b[1,1]<<8) + (b[1,0])))
        bram_b.write(8,  int((b[2,3]<<24) + (b[2,2]<<16) + (b[2,1]<<8) + (b[2,0])))
        bram_b.write(12, int((b[3,3]<<24) + (b[3,2]<<16) + (b[3,1]<<8) + (b[3,0])))

    def read_c(self):
        c = np.ndarray([4,4], dtype=np.uint8)
        for i in range(0,4):
            val = bram_c.read(4*i)
            c[i,0] = ((val & 0x000000ff)>>0)
            c[i,1] = ((val & 0x0000ff00)>>8)
            c[i,2] = ((val & 0x00ff0000)>>16)
            c[i,3] = ((val & 0xff000000)>>24)
        return c
```

Listing 6: Code for the Matmul driver

```
class MatMulDriver(DefaultIP):
    def __init__(self, description):
        super().__init__(description=description)

    bindto = ['xilinx.com:user:matmul:1.0']

    def reset(self):
        matmul.write(0x0,0)
        matmul.write(0x4,0)

    def start(self):
        #trigger adder by writing 1 to "start" register
        matmul.write(0x0,1)

    def is_done(self):
        #read the value in the "done" register
        return matmul.read(0x4)

    def clear_done(self):
        #write 0 to the "start" register to clear it
        matmul.write(0x0, 0)
        #write 1 to the "done" register to clear it
        matmul.write(0x4, 1)

    def current_state(self):
        return matmul.read(0x14)

    def check_sanity(self):
        return hex(matmul.read(0x24))
```

Listing 7: Code for the DMA driver

```
class CDMADriver(DefaultIP):
    def __init__(self, description):
        super().__init__(description=description)

    bindto = ['xilinx.com:ip:axi_cdma:4.1']

    def reset(self):
        dma.register_map.CDMACR = 0x0004

    def do_transfer(self, src_addr, dst_addr, nbytes):
        dma.register_map.CDMACR = 0x0004 #reset the DMA
        dma.register_map.SA = src_addr #set source address
        dma_mmio.write(dma.register_map.DA.address, dst_addr)  #set destination address
        dma.register_map.BTT = nbytes #set number of bytes to transfer and also trigger the DMA
        while (dma.register_map.CDMASR[1]==0): #loop until bit 1 (IDLE) is 0
            pass

    def is_idle(self):
        return (dma.register_map.CDMASR[1]==1)
```

Listing 8: List of registers in the TPU-like design

```

//-----------------------------------------
//Addr 0 : Register with enables for various blocks
//-----------------------------------------
`define REG_ENABLES_ADDR 32'h0
//Bit 0: enable_matmul
//Bit 1: enable_norm
//Bit 2: enable_pool
//Bit 3: enable_activation

//-----------------------------------------
//Addr 4: Register that triggers the whole TPU
//-----------------------------------------
`define REG_STDN_TPU_ADDR 32'h4
//Bit 0: start_tpu
//Bit 31: done_tpu

//-----------------------------------------
//Addr 8: Register that stores the mean of the values
//-----------------------------------------
`define REG_MEAN_ADDR 32'h8
//Bit 7:0: mean

//-----------------------------------------
//Addr A: Register that stores the inverse variance of the values
//-----------------------------------------
`define REG_INV_VAR_ADDR 32'hA
//Bit 7:0: inv_var

//-----------------------------------------
//Addr E: Register that stores the starting address of matrix A in BRAM A
//-----------------------------------------
`define REG_MATRIX_A_ADDR 32'he
//Bit `AWIDTH-1:0 address_mat_a

//-----------------------------------------
//Addr 12: Register that stores the starting address of matrix B in BRAM B
```

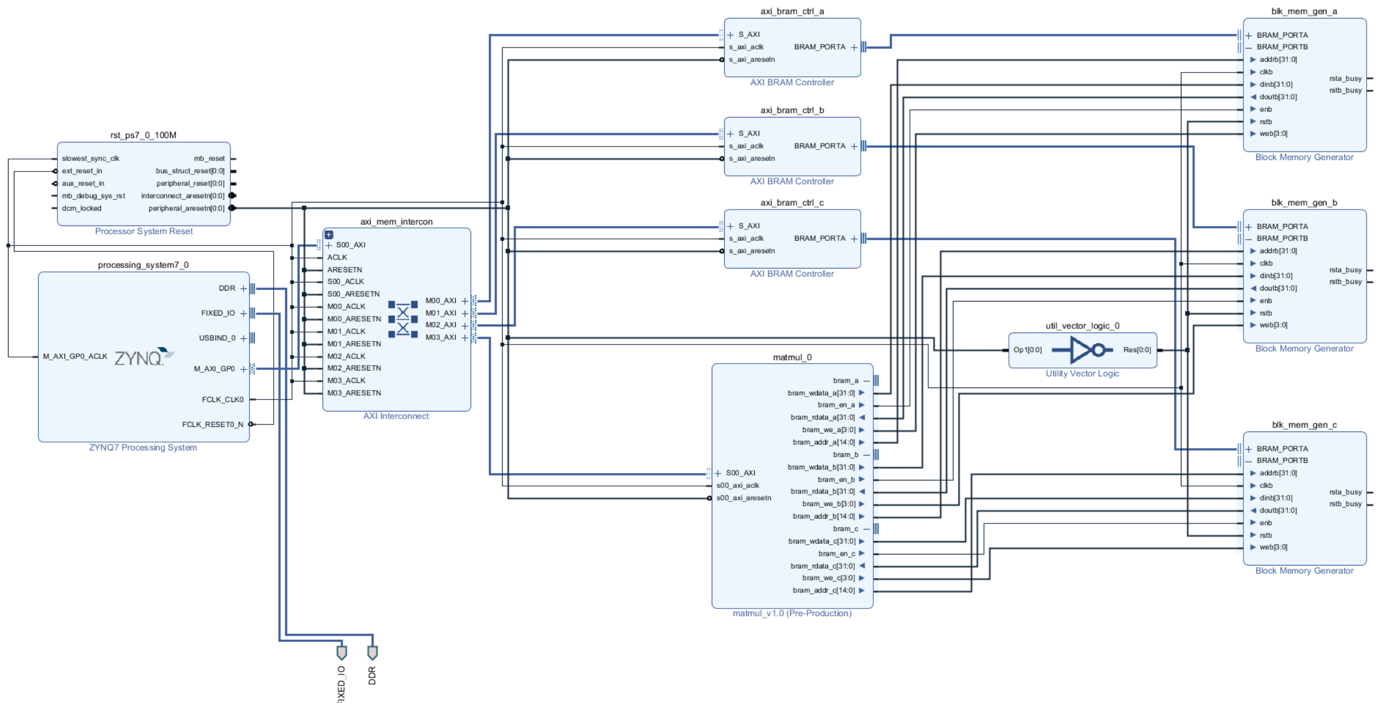

Fig. 12: Snapshot of design overlayed onto the FPGA from Xilinx Vivado. This is the perliminary design that uses MMIO to transfer data to/from the DDR/BRAM

```
38 //----------------------------------------
39 `define REG_MATRIX_B_ADDR 32'h12
40 //Bit `AWIDTH-1:0 address_mat_b
41
42 //----------------------------------------
43 //Addr 16: Register that stores the starting address of matrix C in BRAM C
44 //----------------------------------------
45 `define REG_MATRIX_C_ADDR 32'h16
46 //Bit `AWIDTH-1:0 address_mat_c
47
48 //----------------------------------------
49 //Addr 20: Register that stores the mask of which parts of the matrices are valid.
50 //
51 //Some examples where this is useful:
52 //1. Input matrix is smaller than the matmul.
53 //   Say we want to multiply a 6x6 using an 8x8 matmul.
54 //   The matmul still operates on the whole 8x8 part, so we need
55 //   to ensure that there are 0s in the BRAMs in the invalid parts.
56 //   But the mask is used by the blocks other than matmul. For ex,
57 //   norm block will use the mask to avoid applying mean and variance
58 //   to invalid parts (so tha they stay 0).
59 //2. When we start with large matrices, the size of the matrices can
60 //   reduce to something less than the matmul size because of pooling.
61 //   In that case for the next layer, we need to tell blocks like norm,
62 //   what is valid and what is not.
63 //
64 //Note: This masks is applied to both x and y directions and also
65 //applied to both input matrices - A and B.
66 //----------------------------------------
67 `define REG_VALID_MASK_ADDR 32'h20
68 //Bit `MASK_WIDTH-1:0 mask for a row or column of both matrices
69
70 //----------------------------------------
71 //Addr 24: Register that controls the accumulation logic
72 //----------------------------------------
73 `define REG_ACCUM_ACTIONS_ADDR 32'h24
74 //Bit 0 save_output_to_accumulator
75 //Bit 1 add_accumulator_to_output
76
77 //----------------------------------------
78 //Addr 28: Register that stores the stride that should be taken to address
79 //elements in matrix A, after every MAT_MUL_SIZE worth of data has been fetched.
80 //See the diagram in "Meeting-16" notes in the EE382V project Onenote notebook.
81 //This stride is applied when incrementing addresses for matrix A in the vertical
82 //direction.
83 //----------------------------------------
84 `define REG_MATRIX_A_STRIDE_ADDR 32'h28
85 //Bit `ADDR_STRIDE_WIDTH-1:0 address_stride_a
86
87 //----------------------------------------
88 //Addr 32: Register that stores the stride that should be taken to address
89 //elements in matrix B, after every MAT_MUL_SIZE worth of data has been fetched.
90 //See the diagram in "Meeting-16" notes in the EE382V project Onenote notebook.
91 //This stride is applied when incrementing addresses
```

Fig. 13: Snapshot of design overlayed onto the FPGA from Xilinx Vivado. This is the final design that uses a DMA engine to transfer data to/from the DDR/BRAM

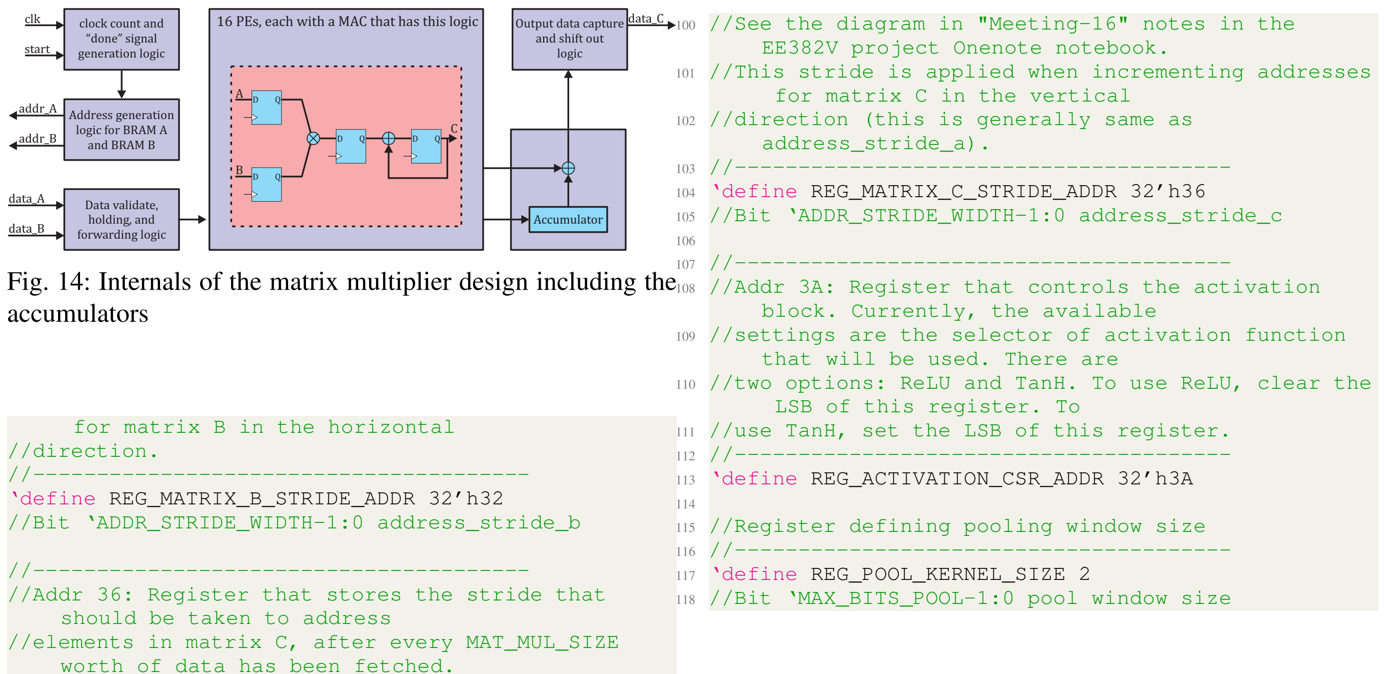

Fig. 14: Internals of the matrix multiplier design including the accumulators

```
    for matrix B in the horizontal
92 //direction.
93 //---------------------------------------
94 `define REG_MATRIX_B_STRIDE_ADDR 32'h32
95 //Bit `ADDR_STRIDE_WIDTH-1:0 address_stride_b
96
97 //---------------------------------------
98 //Addr 36: Register that stores the stride that
       should be taken to address
99 //elements in matrix C, after every MAT_MUL_SIZE
       worth of data has been fetched.
100 //See the diagram in "Meeting-16" notes in the
       EE382V project Onenote notebook.
101 //This stride is applied when incrementing addresses
        for matrix C in the vertical
102 //direction (this is generally same as
       address_stride_a).
103 //---------------------------------------
104 `define REG_MATRIX_C_STRIDE_ADDR 32'h36
105 //Bit `ADDR_STRIDE_WIDTH-1:0 address_stride_c
106
107 //---------------------------------------
108 //Addr 3A: Register that controls the activation
       block. Currently, the available
109 //settings are the selector of activation function
       that will be used. There are
110 //two options: ReLU and TanH. To use ReLU, clear the
        LSB of this register. To
111 //use TanH, set the LSB of this register.
112 //---------------------------------------
113 `define REG_ACTIVATION_CSR_ADDR 32'h3A
114
115 //Register defining pooling window size
116 //---------------------------------------
117 `define REG_POOL_KERNEL_SIZE 2
118 //Bit `MAX_BITS_POOL-1:0 pool window size
```

Fig. 15: Waveforms from simulation of the TPU-design

Fig. 16: Layout generated by VTR for the TPU-like design using the Stratix-IV like architecture (Blue=LBs, Yellow=BRAMs, Pink=DSP slices)