tinyVTA: General Purpose FPGA-based Tensor Accelerator for Deep Neural Networks

Sankalp Bhamare[1], Rohan Rajesh Kalbag[2], and Rishabh Ravi[3]

[1]**200110096@iitb.ac.in, B.Tech 4th Year, Electrical Engineering, Indian Institute of Technology Bombay**
[2]**20d170033@iitb.ac.in, B.Tech 4th Year, Electrical Engineering, Indian Institute of Technology Bombay**
[3]**200260041@iitb.ac.in, DD 4th Year, Electrical Engineering, Indian Institute of Technology Bombay**

ABSTRACT

We implement a system which accelerates expensive floating-point operations such as block matrix-multiply-accumulate (MMAC) and block activation function application (ACTIV) operations of Deep Neural Network inference to an Xilinx UltraScale+ based FPGA. Additionally, we've developed a simple compiler for the accelerator that compiles sequential models from the ML framework (e.g., PyTorch) into instructions and data memory, which can be seamlessly loaded onto the accelerator for inference.

Keywords: Deep Learning, Field Programmable Gate Array, Hardware Acceleration

1 ACCELERATOR ARCHITECTURE

The accelerator consists of specialized kernels which perform generalized block matrix multiply-accumulate (MMAC) and block activation application (ACTIV), instruction and data memory ports [1], and control logic. In the case of our implementation of tinyVTA, Vitis HLS was used to develop the individual components. We go over these individual components in detail below:

1.1 Block Matrix Multiply-Accumulate (MMAC)

This kernel is connected to the data memory via the m_axi interface, and takes start base offsets for each of the input matrices bA and bB, and the start offset for the result matrix bAB, Block Count (N) as parameterization i.e. takes $(N*BLOCK\_SIZE) \times (N*BLOCK\_SIZE)$ Matrix as input.

Its functionality is described as follows, it reads the three matrices A, B, AB corresponding to the addresses $[bA : bA + (N*BLOCK\_SIZE) \times (N*BLOCK\_SIZE)]$, $[bB : bB + (N*BLOCK\_SIZE) \times (N*BLOCK\_SIZE]$ and $[bAB : bAB + (N*BLOCK\_SIZE) \times (N*BLOCK\_SIZE)]$ from the data memory and performs the operation and stores the result back in AB

$$AB \leftarrow (A \times B) + AB$$

The blockwise partitioning and mapping of the $\times$ operation is described in Figure 1

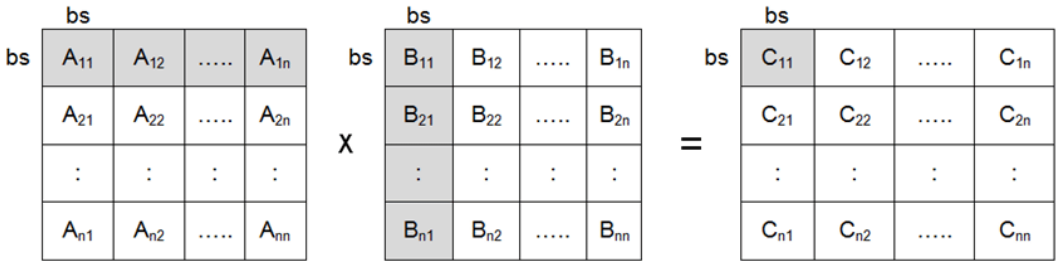

Figure 1. Block Matrix Multiplication

The HLS kernel signature for MMAC is like (here `CT` is `hls::vector<float,BLOCK_SIZE>`):

```
void matmul(CT *mem, int bA, int bB, int bAB, int N)
```

[1]Connected through m_axi interface to UltraScale+ PS Buffers

1.2 Block Activation Application (ACTIV)

This kernel again is connected to the data memory by means of the m_axi interface, and takes as input Block Count (N) which corresponds to the number of blocks of the matrix, the base address *ro* and *wo* corresponding to the input and output matrices.

Its functionality is as follows, it reads the matrix A corresonding to the address $[ro : ro + (N * BLOCK\_SIZE)]$ and for each element of this matrix it applies elementwise the activation function f_{activ} which can be ReLU, tanh, sigmoid

The HLS kernel signature for ACTIV is like (here `CT` is `hls::vector<float,BLOCK_SIZE>`):

```
void activ(CT *A, int ro, int wo, int N)
```

1.3 Data Memory

As discussed earlier the data memory port is described using the datatype `CT`, an alias for `hls::vector<float,BLOCK_SIZE>`. This is connected by means of m_axi to PynqBuffer objects (which behave similar to numpy arrays of floats) allocated on the UltraScale+ processor.

1.4 Instruction Memory

The instruction memory port is described using the datatype `inst_t` which is an alias for `ap_int<64>` which corresponds to a bitvector represented by a 64-bit integer. This is again connected by means of m_axi to PynqBuffer object (similar to numpy arrays with 64-bit integers) allocated on the UltraScale+ processor.

The data memory and instruction memory buffers can be read from and written to, by means of the pythonic interface Pynq for UltraScale+ offers, the kernels and interact with them in these PynqBuffer objects by means of AXI4 transfer in burst mode.

1.5 Instruction Set Architecture (ISA)

The control logic must expose an instruction set which allows the software to describe the invocations of the MMAC and ACTIV kernels, specify address offsets and block sizes. To do the following the following ISA was selected for the accelerator:

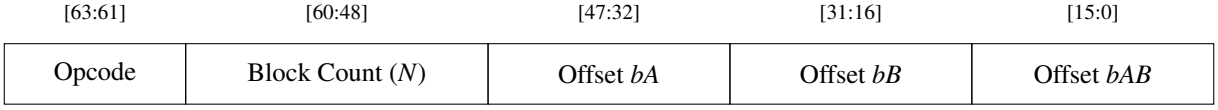

Figure 2. Instruction Set Architecture

- **Opcode** : This can take two values `001` and `010` corresponding to ACTIV and MMAC respectively, this selects which the kernel operation should the control logic process now.
- **Block Count (N)** : This is used to feed the value of N (block count, i.e number of blocks to be considered after the base offsets provided to the kernel operations)
- **Offset** *bA*: This is used to feed to value of *bA* to MMAC, and *ro* for ACTIV
- **Offset** *bB*: This is used to feed to value of *bB* to MMAC, and *wo* for ACTIV
- **Offset** *bAB*: This is used to feed to value of *bAB* to MMAC, and doesn't matter (consists of don't care values) for ACTIV since it only requires two base offset addresses.

1.6 Control Logic

This is the main processing logic that interfaces with all the individual components of the architecture. It consists of a program counter initialized to zero. The logic reads each instruction from the instruction memory, decodes it according to the ISA shown above, and feeds the corresponding inputs to the kernels. After processing the current instruction, it advances to the next instruction by incrementing the program counter. This process continues until the instruction is entirely zero, indicated by `00..00`, which signals the end of computation.

1.7 Block Design

The PS/PL Interface Block design consists of **Zynq UltraScale+ MPSoC** connected to our **acti_proc** HLS IP [2] using AXI SmartConnect. The m_axi ports of the IP are connected to the PS/PL via the AXI HPO FPD and AXI HP1 FPD ports enabled on the SoC to enable DMA connection.

[2] Implementation of the proposed accelerator in Vitis.

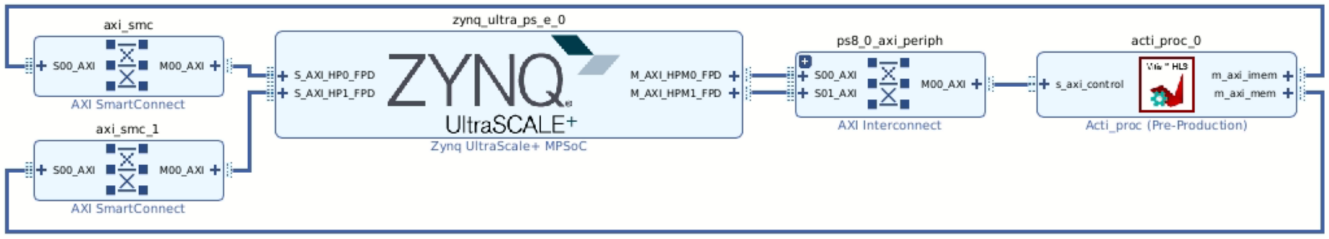

Figure 3. Block Design

2 SOFTWARE UTILITIES FOR tinyVTA ARCHITECTURE

The tinyVTA accelerator architecture can be used for the inference of fully-connected neural network model architectures with **ReLU**[3] activation which can be described using ML frameworks such as **PyTorch** or **Tensorflow**[4]. After training on the desired data offline, all the model weights W_i and biases B_i corresponding to the i^{th} layer ($1 \leq i \leq N$, where N is the number of layers present in the fully-connected neural network model) are learned and extracted from the model.

2.1 Preprocessing and Padding Weights and Biases

Since tinyVTA implements block multiplication for a predetermined fixed *BLOCK_SIZE* (currently chosen as 16), it can only perform MMAC for square matrices with their dimension which is a multiple of this *BLOCK_SIZE*. This requires transformation of the inputs, weights and biases to supported sizes by means of zero padding. We implement a utility which appropriately pads them as follows:

- For the inputs $x_i \in \mathbb{R}^{d_1}$ (input has d_1 features), at once we can perform the inference of t examples where $t \leq BLOCK\_SIZE$. The utility finds the smallest d_1', a multiple of *BLOCK_SIZE* such that $d_1' \geq d_1$, and creates $x_i' \in \mathbb{R}^{d_1'}$ such that x_i occupies the left corner and rest of the elements are zeroes, all the padded inference examples are made the first t rows of a matrix $X \in \mathbb{R}^{BLOCK\_SIZE \times d_1'}$.
- For weights matrices $W_i \in \mathbb{R}^{d_1 \times d_2}$, the utility finds the smallest d_1' and d_2' which are multiples of *BLOCK_SIZE* such that $d_1' \geq d_1$, $d_2' \geq d_2$ and creates $W_i' \in \mathbb{R}^{d_1' \times d_2'}$ such that W_i occupies the top left corner of W_i' and rest of the elements are zeroes.
- For bias vector $B_i \in \mathbb{R}^{d_2}$, the utility finds the smallest d_1', a multiple of *BLOCK_SIZE* such that $d_1' \geq d_1$, and creates $B_i^t \in \mathbb{R}^{d_1'}$ such that B_i occupies the left corner of B_i^t and rest of the elements are zeroes. The bias vector is replicated *BLOCK_SIZE* number of times to account for the t input examples to give B_i' whose each row is B_i^t.

2.2 Neural Network Inference

The inference operation of the fully connected network, in terms of the above two operations **MMAC** and **ACTIV** exposed by tinyVTA is described in Algorithm 1

Algorithm 1 Model Inference expressed using MMAC and ACTIV

Input: Padded Model Weights W_i', Padded Model Biases B_i' $1 \leq i \leq N$, Padded Model input X
Output: Model Prediction y_{pred}

$acc \leftarrow x_{inp}$
for $i \leftarrow 1$ to $N-1$ **do**
  $acc \leftarrow acc \times W_i + B_i$
  $acc \leftarrow f_{activ}(acc)$
end for
$acc \leftarrow acc \times W_N + B_N$
$y_{pred} \leftarrow acc$

[3]The design can be extended to other activation functions like sigmoid, tanh
[4]Other frameworks can also be supported as long as the weights and biases are exported into numpy arrays

2.3 Compiler

We have designed a compiler which given a model utilizes all the padding utilities and gets the padded weights, inputs and biases and generates the data memory and offsets for each of these matrices, it then implements Algorithm 1 and translates all the $acc \leftarrow acc \times W + B$ operations to MMAC instructions, $acc \leftarrow f_{activ}(acc)$ to ACTIV instructions and identifies the offsets, block count corresponding each matrix in the data memory. Using these offsets, N's and instructions, it generates tinyVTA ISA instructions which constitute the instruction memory.

3 DOCUMENTATION AND USAGE

3.1 FPGA Side

The `RPCServer` directory containing all FPGA side software consists of the following:

- A `dma_ps` folder which must contain the bitstream, hardware handoff and tcl script generated by Vivado after block design of Accelerator IP. Currently they are named as `design_1.bit`, `design_1.hwh`, and `design_1.tcl`
- The RPC server side python script `rpc_server.py` which contains our custom implementation of an RPC server implemented using tornado which exposes an endpoint `/program` to which the RPC client issues POST requests to

3.1.1 Usage

To set up the FPGA side software do the following:

- Run `sudo -i` and enter as root user by entering password `xilinx`
- Start the RPC server using `python3 rpc_server.py` and wait for `INFO: RPC server is up and running` log message to before starting issuing any client-side requests

3.2 Client Side

The `TinyTVM` directory contains the client-side software which consists of the following:

- `pack_memory.py` which implements the padding strategies for model inputs, weights and biases as mentioned in Section 2.1.
- `compiler.py` which contains the implementation of the compiler discussed is Section 2.3
- `utils.py` containing the padding and other compiler helper functions
- `rpc_client.py` which contains our custom implementation of the RPC Client which makes POST requests to the RPC Server on the FPGA, it sends a payload containing the instruction memory and data memory and performs the inference on the FPGA and returns the predicted inference output in the return HTTP response. It contains a `UltraScaleRPCClient` class which takes in the server ip and port corresponding to the RPC server. A request to perform inference can be done using the function `perform_inference(model, input)` where the model object and inputs to the model whose inference need to be done

3.2.1 Usage

An example code snippet describing the invocation client side is below, for more details refer to the example ipython notebook provided.

```
usrpc = rpc_client.UltraScaleRPCClient(<zcu104_ip>, port)
y_hw = usrpc.perform_inference(model, test_x[:128])
```

4 TESTING AND VERIFICATION

We evaluated the correctness of tinyVTA using a large fully connected model with 26,122 parameters (as shown in Figure 5), trained to perform 8x8 MNIST digit classification, the model consists of 3 linear layers of dimensions (input, output) : $(64, 128)$, $(128, 128)$, and $(128, 10)$, each followed by a **ReLU** activation layer except for the final layer output which is fed to a **softmax** activation to get probabilities of labels as depicted in Figure 4. To verify the implementation, we conducted both software inference and hardware inference and compared the results.

Our compiler was used to generate the data memory and the instructions for the model. The instructions obtained for this particular model when assembled were the following, correctly consisting of 3 MMAC instructions and and two ReLU activations.

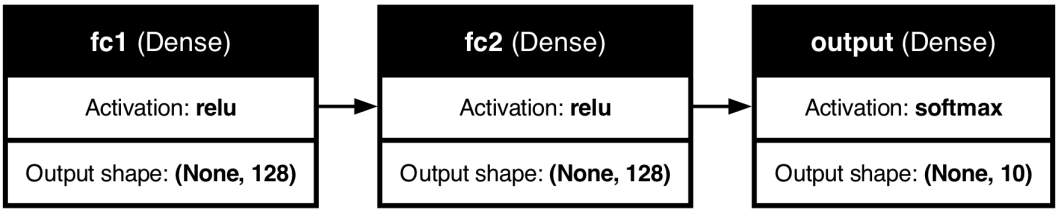

Figure 4. 8x8 MNIST Model Architecture

Model: "sequential_5"

Layer (type)	Output Shape	Param #
fc1 (Dense)	(None, 128)	8,320
fc2 (Dense)	(None, 128)	16,512
output (Dense)	(None, 10)	1,290

Total params: 26,122 (102.04 KB)
Trainable params: 26,122 (102.04 KB)
Non-trainable params: 0 (0.00 B)

Figure 5. The number of trainable parameters in the 64-128-128-10 model chosen for MNIST inference

```
MMAC 8, 0x0, 0x400, 0x1000
ACTIV 1024, 0x1000, 0x1000, 0x0
MMAC 8, 0x1000, 0x800, 0x1400
ACTIV 1024, 0x1400, 0x1400, 0x0
MMAC 8, 0x1400, 0xc00, 0x1800
```

The final output is then read from the address location `0x1800` onwards and np.argmax is done in software to identify the predicted class.

4.1 Software Testbench based Design Verification

The software testbench designed to verify our hardware on **Vitis CSIM** involves reading the data and instructions from text files, then simulating the accelerator IP connected to the data and instruction memory and finally, verifying the correctness of the evaluated output values against the expected output.

4.2 Hardware Design Verification

We employed the **UltraScale+ Overlay** to establish a robust software interface for our accelerator implementation IP, denoted as **acti_proc**. This overlay instantiation took place within the **Xilinx ZCU104 FPGA** at the HPC Lab. To facilitate communication with the accelerator, we created PynqBuffer objects for both instruction memory and data memory. Subsequently, these buffers were populated with the requisite instructions and data memory content, which were generated by our compiler.

Following the setup, we initiated the start signal to activate the overlay, enabling the execution of the model. The resultant outputs from the model were retrieved from the data memory PynqBuffer. We then conducted an argmax operation on these outputs, thereby obtaining the corresponding labels. To validate the accuracy of our accelerator design, we compared these labels with those obtained through pure software inference performed in Python.

Remarkably, the results were consistent across all 128 test examples, as demonstrated in Figure 6, affirming the correctness of our accelerator design.

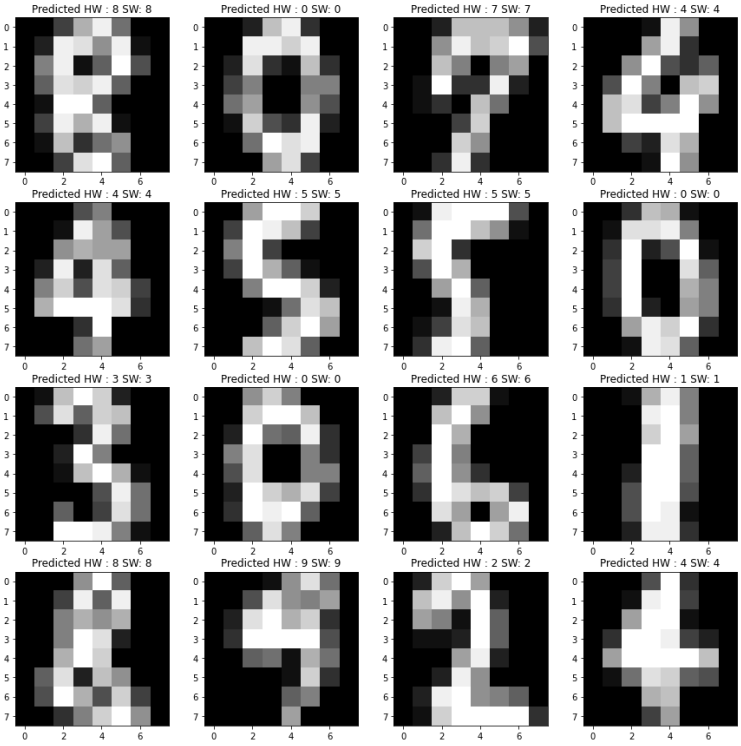

Figure 6. Some input examples dispatched to tinyVTA for inference visualized with their HW predicted labels and SW predicted labels in the title of each subplot which are matching in all of the cases

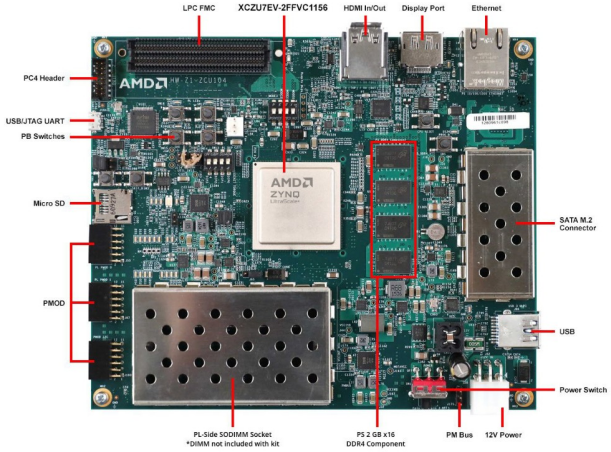

Figure 7. ZCU104 Board

ACKNOWLEDGMENTS

We would like to thank **Prof. Sachin Patkar**, Department of Electrical Engineering, IIT Bombay for being a source of guidance and for providing us the opportunity and resources to work on this project.