Hardware Accelerator for a Simple Neural Network

Shashank Nag
EE19B118
IIT Madras

Sai Gautham Ravipati
EE19B053
IIT Madras

Vishnu Varma V
EE19B059
IIT Madras

1 Goal of the Project

The goal of this project is to analyse and accelerate the task of predicting the number on an MNIST image using a pre-trained neural network. We would analyse the case of performing the entire computation on software in the PicoRV32 processor to identify the primary bottlenecks. Based on this analysis, an accelerator would be designed, analysed and interfaced, aiming to improve overall performance.

2 Baseline Code

The neural network architecture (in Fig. 1) with two layers and 32-hidden units has been used for the purpose. The input is a (28 x 28) MNIST image which is re-shaped to (784 x 1). The output is the scores for the 10 categories of integers from [0-9], with a higher score signifying greater probability of that image. The weights are generated using Tensor-flow, and post training quantisation is performed to convert the weights to integers, since it would be easier for analysis on PicoRV32 which supports only integer operations. The forward pass to predict the class of a test image using these weights is implemented in C using for-loops, corresponding to $O(N^2)$ algorithm for matrix multiplication (hello.c can be referred for the same). Weights (W1) and bias (b1) for the first layer, weights (W2) and bias (b2) for the second layer, and the input vector are stored in the file 'wbd.hex', and the same is accessed by the C code to load data. The C code could have been coded in a way such that there shall be better data re-use, say using implementations like blocked matrix multiplication or loop unrolling, but there would be no associated speed up on a cache-less PicoRV32 processor, as such speed-ups are achieved by exploiting temporal and spatial locality of reference of the cache.

```
Sequential
(
  (0): Linear(in_features = 784, out_features = 032, bias=True)
  (1): ReLU()
  (2): Linear(in_features = 032, out_features = 010, bias=True)
)
```

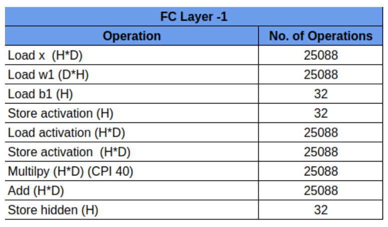
Figure 1: Neural Network Architecture

Analysis of the baseline code

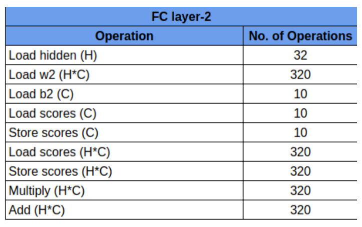
The forward pass involves three stages - first stage involves matrix multiplication (1x784) x (784x32) to get a set of hidden layer units (1x32), the second stage being ReLU involves logical operations, and the third stage involves another matrix multiplication (1x32) x (1x10). We infer the class as the one with the maximum score. The forward pass of the naive implementation took 12350109 cycles on the PicoRV32.

Instrumenting the code, we find that each stage takes the following number of cycles - FC layer-1 (~12.8 M cycles), ReLU (480 cycles), FC layer-2 (~0.16 M cycles). It is evident that of the 3 stages, FC layer-1 takes the largest number of cycles. Being the largest matrix multiplication stage, it takes the most number of cycles, while ReLU being a logic based computation is performed fairly efficiently on software.

To analyse on the instructions that go behind the matrix multiplication stage, we resort to a theoretical approach, assuming each element is accessed from memory in each computation it is involved.



FC Layer -1	
Operation	No. of Operations
Load x (H*D)	25088
Load w1 (D*H)	25088
Load b1 (H)	32
Store activation (H)	32
Load activation (H*D)	25088
Store activation (H*D)	25088
Multilpy (H*D) (CPI 40)	25088
Add (H*D)	25088
Store hidden (H)	32



FC layer-2	
Operation	No. of Operations
Load hidden (H)	32
Load w2 (H*C)	320
Load b2 (C)	10
Load scores (C)	10
Store scores (C)	10
Load scores (H*C)	320
Store scores (H*C)	320
Multiply (H*C)	320
Add (H*C)	320

Figure 2: Analysing theoretical number of operations for baseline code

Based on this, we assert that the actual computation of the matrix vector multiplication is the dampener when compared to the ReLU and this is owing to the very sequential nature of execution of the C code. Further each element had to be loaded more than once from memory, unlike performing parallel operations, and this takes significant time, while this can be optimised used parallel reads with BRAM's on a FPGA and at the same time PicoRV32 implements sequential multiplication and the same can be optimised by the usage of DSP blocks in a FPGA, which implement combinational multiplication. Thus, we hypothesise that such tasks like matrix multiplications could be performed more efficiently on hardware accelerators due to their inherent parallelism.

3 Hardware Implementation

Based on the above hypothesis, we turn to implementing the matrix multiplication on hardware. Hardware matrix multipliers have been the subject of extensive research, with state of the architectures like Google TPU delivering a high performance. Most of these are based on the systolic array architecture, with N^2 multiply-accumulate (MAC) cells for multiplying two NXN matrices.

Upon a closer analysis, we find two key attributes of these designs that require attention while implementing in our project. One, the architectures assume that the data required for computation are all readily available to the peripheral upon request, and these are read into the MAC cells in a pipelined fashion. Two, the architecture is not directly scalable, as it would involve several MAC cells to compute the product. The latter could be tackled by computing the products taking block matrices one at a time. For our case, we go ahead with the entire matrix as we would require only 32 MAC cells for the first stage, and 10 for the third stage.

We could go for a fully parallel implementation computing all the required products in one go and accumulating those, but the primary bottleneck noticed was in getting the data for first matrix (image) and the second matrix (weights). Receiving data from the core (PicoRV32) takes multiple clock cycles after a request, and assuming that we don't change the bus, each of this has to be done sequentially. Hence, pipelining offers no advantage. To reduce a two fold wait to get data from the core, we decided to load the weights for the matrix multiplication onto the memory of the accelerator itself (BRAMs incase of FPGA). Further, to ensure the best utilisation of parallelism on hardware, we decided to store the different columns of the weight matrix in different BRAMs so that they could be accessed parallely (column major). This itself provides a very significant speedup compared to the baseline case where it was processing each product sequentially.

Though our initial hypothesis was to implement the matrix multiplier unit alone in hardware, we figured out that the primary bottleneck was loading data onto the peripheral from the core and writing back. Hence, doing this for the two matrix multiplication stages and performing the other computations on the software would be meaningless due to the overhead of sending and writing back data. Based on the fact that a ReLU activation can be efficiently implemented on hardware using a comparator and a MUX, we decided to implement the entire neural network as a single accelerator unit. Further, to reduce the number of stages, we decided to augment the bias term as well into MAC operations, by appending the bias to the last row of the weights matrix, and padding 1 to the end of the image.

Based on these analysis, we came up with the design for the accelerator as in (Fig. 2). The image pixels (784 in number) are sent sequentially by the core through the bus, and the accelerator performs the set of computations and returns the scores for the 10 classes at the end of computations. We have taken care to ensure that there are no overflows in the process. We used appropriate counters and start/stop signals for each of the processing units, and a ready signal that signals end of computation to the core.

We were faced with some more design choices over the course of this process. One of the key being the choice of a sequential and combinational multiplier. The latter would take fewer number of clock cycles, but could lead to worse critical paths. However, as our design uses 42 MAC units, we could easily implement these multipliers on DSP slices on FPGAs which have a decent delay. We decided to go ahead with combinational multipliers and analysed the same subsequently. Since we had loaded in all the image pixels onto the registers, we had another choice of loading in all weights into registers and processing these as well parallely. But we decided to drop this due to the humongous amount of MAC units and registers required in that case.

Analysing the Implementation

The hardware accelerator was interfaced with the core and we analysed the performance of the same. We obtained the benchmarks for the different parts of the accelerator and compared the overall performance with the baseline implementation. We have used a Kintex-7 series target FPGA for performing the timing related analysis.

- The accelerator is implemented such that the core computations of the 3 stages of the neural network is completed in 840 cycles (800 cycles for the 1st stage + combinational ReLU + 40 cycles for the 3rd stage). This includes a buffer of a few cycles to ensure that the results stabilise.
- As expected, the key bottleneck is transfer of data to and from the peripheral. Writing the image to the peripheral and allied processes take about $20,048$ cycles.
- Post this, the core receives a ready signal from 924 cycles, and takes about $28,000$ cycles to get the results and print them.

The above numbers justify our decision of implementing the entire process on the peripheral rather than the matrix multiplication unit alone. If we were to do so, we would have almost doubled the cycles taken here, as the key bottleneck is data transfer. By using multiple BRAMs for weights (in column major form), we have also eliminated the multi cycle wait for the same.

4 Results & Conclusions

A direct comparison with the baseline for this would not be meaningful as we might have altered the critical path delay while building the accelerator with MAC cells. On analysis, it indeed is the case. The benchmarks for PicoRV32 for a Kintex-7T FPGA indicate an optimum clock period of 2.2 ns. Going by a worst case assumption, we find that timing constraints on the same device are met for the accelerator for 8.8 ns clock period. We performed the speedup analysis assuming the design now runs at this new clock period.

- Overall Speedup : The baseline code took 12379773 cycles for the computation, while the code after interfacing the accelerator took about 48216 cycles. Taking into account the 4x change in clock period, this gives an overall speedup of 64x.
- As expected, the key bottleneck is transfer of data to and from the peripheral. Writing the image to the peripheral and allied processes take about $20,048$ cycles.
- Post this, the core receives a ready signal from 924 cycles, and takes about $28,000$ cycles to get the results and print them.
- As noted, the overhead of sending and receiving data to the peripheral is much greater compared to the actual compute time of 840 clock cycles. This suggests we could further improve the design by using a high bandwidth bus.
- The multiple cycles taken to send data is because of the fact that the pointer is updated by the software, which happens only in several cycles. A possible modification that could be done to the bus in PicoRV32 is that it sends out image pixels sequentially every cycle once the initial pointer is addressed. If this is possible, we could pipeline the implementation and complete the entire operation about 1000 cycles.

As noted above, the accelerator worsens the critical path, thus affecting the operable frequency of the core as well. This could be easily worked around, by operating the accelerator at a different clock frequency, thus allowing the core to operate as usual. Another possibility could be to use a sequential multiplier, which would take larger number of clock cycles, but might not worsen the critical path as much as this case. Based on the analysis above, we conclude that the hardware accelerator provides an extremely significant speedup in computing about 64 times. The main factors of speed-up are due to parallel computation using the MACs as well as the parallel data reading of the weights, which wasn't the case of the CPU and the same was being done sequentially.

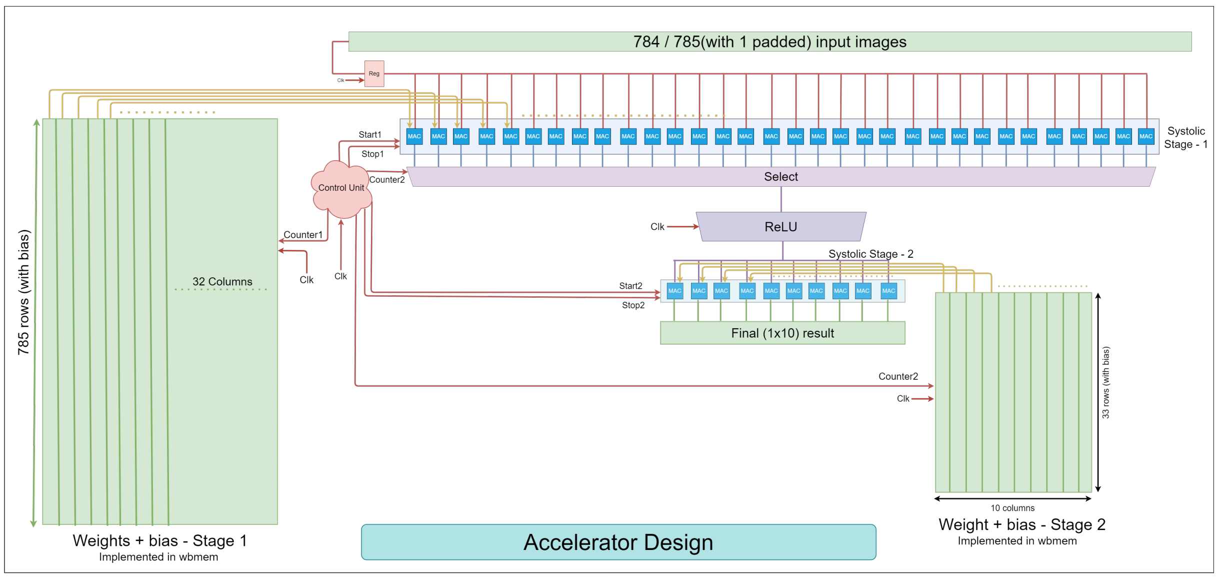


Figure 3: Accelerator Design

5 Work Distribution

- Sai Gautham Ravipati, EE19B053 - Base-line code (hello.c and data files), ideation of accelerator architecture (primitive), verilog code of accelerator (xmem.v, wbmem.v, .mem files), documentation of code, de-bugging of verilog code for accelerator and integrated peripheral, synthesis of accelerator in yosys. report (Goal of the Project, Baseline Code)
- Shashank Nag, EE19B118 - Ideation of accelerator architecture (final), verilog code for accelerator (except wbmem.v, systolic2.v, xmem.v), de-bugging the code for accelerator and integrated peripheral, cycle count for stages for base-line code, contributed to Xilinx synthesis, report (Baseline Code, Hardware Implementation)
- Vishnu Varma V, EE19B059 - Verilog code for accelerator (systolic2.v), minor contribution in other accelerator modules, integration of accelerator with PicoRV32 (axi-mem-periph.v, hello.c) and interfacing with accelerator.v, synthesis of accelerator and timing report in Xilinx. report (Accelerator design - Image, Hardware Implementation)

6 Code

The code for this project could be found `https://git.ee2003.dev.iitm.ac.in/ee19b118/Course_Project`