2019 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION INDIA

Hardware implementation of smallscale parameterized neural network inference engine

Vishnu P Bharadwaj, Logic Design Engineer, INTEL, Bengaluru, India (*vishnu.bharadwaj@intel.com*)

Shruti Narake, Logic Design Engineer, INTEL, Bengaluru, India (*shruti.narake@intel.com*)

Saurabh D Patil, Logic Design Engineer, Affiliation, INTEL, Bengaluru, India (*saurabh.d.patil@intel.com*)

***Abstract*—This paper presents a scalable and parameterized hardware architecture for convolutional and fully connected neural networks. This is a quick and low cost inference engine for verifying solutions in hardware. Architecture with four convolution operations in parallel with parallel multiplier and adder are implemented for increased performance and larger timing margin. The architecture is easily customizable for many topologies with minimal changes. Modular architecture with parameterized design provides an excellent workbench for fast and easy neural network applications.**

Keywords—Artificial intelligence; convolutional neural networks; fully connected layer; machine learning; pipelined architecture;

I. INTRODUCTION

The advancements in VLSI technology allow us to take the advantage of offloading the computationally intensive tasks onto the dedicated hardware and thereby letting the main processor free for performing other important tasks. It is also desirable that electronic devices are capable of sensing and understanding their surroundings and adapting their services according to the context. Artificial Neural Networks (ANN) have been in spot-light for this purpose, primarily due to their wide range of applicability. The Multilayer Perceptron (MLP) is the most frequently used ANN due to its ability to model non-linear systems and establish non-linear decision boundaries in classification problems such as OCR, data mining and image processing/recognition. However, since MLP requires extremely high throughput, this computational complexity is highly undesirable from real time operations for embedded devices which have constraints in their processing capabilities. An attractive solution to this is to design a dedicated hardware for MLP acceleration.

Related works are FPGA based ConvNets processor [1] which speaks about implementation of convolutional neural networks in low-end FPGAs with external memory with help of network compiler software to implement instructions. Convolutions using Winograd engine [2] was used in implementing complex topologies in FPGA for higher performance and parallel operation alongside host processor. This paper implements the architecture relating the two i.e. simpler topology for minimal power and gate count consumption and reusable hardware to support both convolutional neural networks (CNN) and fully connected layer (FCL). We implement a parameterized and scalable architecture with pipelined operations for higher throughput and large timing margins.

II. SOFTWARE IMPLEMENTATION

Prior to hardware implementation, the design was modelled in software as a golden reference to compare outputs. Few changes in software implementation to ease the verification in hardware was also carried out as depicted in next section.

Google's Tensorflow framework was chosen for software implementation of neural network topology due to its ease of configuration and display of trained data. Tensorflow™ is an open source software library for numerical computation using data flow graphs [3]. It provides APIs for training and inference with new algorithms for maximum accuracy. To keep software implementation simple and eliminate noise, reliable and large dataset of MNIST was used. The large datasets in short have a good quality signal to noise ratio overall. LENET topology was implemented in tensorflow and trained to obtain the high accuracy i.e. >95%.

Flow diagram of implementation to validate hardware results is given in figure 1. This design was tweaked to ease hardware verification. As all operations are floating point in tensorflow compute elements are implemented to support integer arithmetic operations. Trained weights were tweaked to have only positive integers. The conversion is performed by rounding of these weights after adding 0.5 to the existing decimal values. Python's numpy package is used to perform these operations.

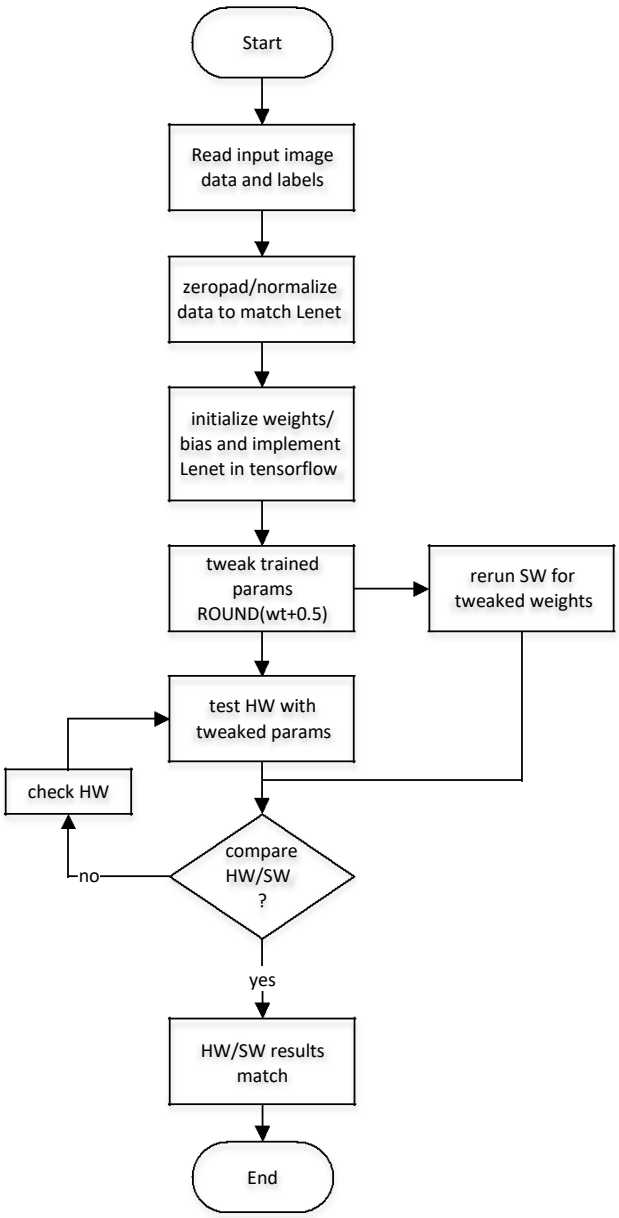

Figure 1. Implementation flowchart

III. HARDWARE IMPLEMENTATION

A. CNN implemetation

A High level architecture of hardware implementation is given in below figure 2. Image pixels for convolution are read from memory and provided to line buffer where the length of the buffer is configurable depending on size and stride of convolution. When start of frame (SOF) signal asserts, a counter starts counting till it reaches the end of the buffer (yellow block in figure). Once SOF reaches the end, 4 pixels from each output of buffer from specified positions are taken as input data for convolution and buffered once. In next cycle, another set of 4 pixels are taken and combined with previous pixel making an 8 pixel column. These 8 pixel with 5 rows are used for convolution. Below image depicts five outputs from line buffer meaning 5x5 filter convolution. With 8 pixel and stride of one, four simultaneous operations can be done. These data are fed to compute block through a routing block which

provides the data in the required format. This is done by swizzling the data to output in matrix format for MAC operations.

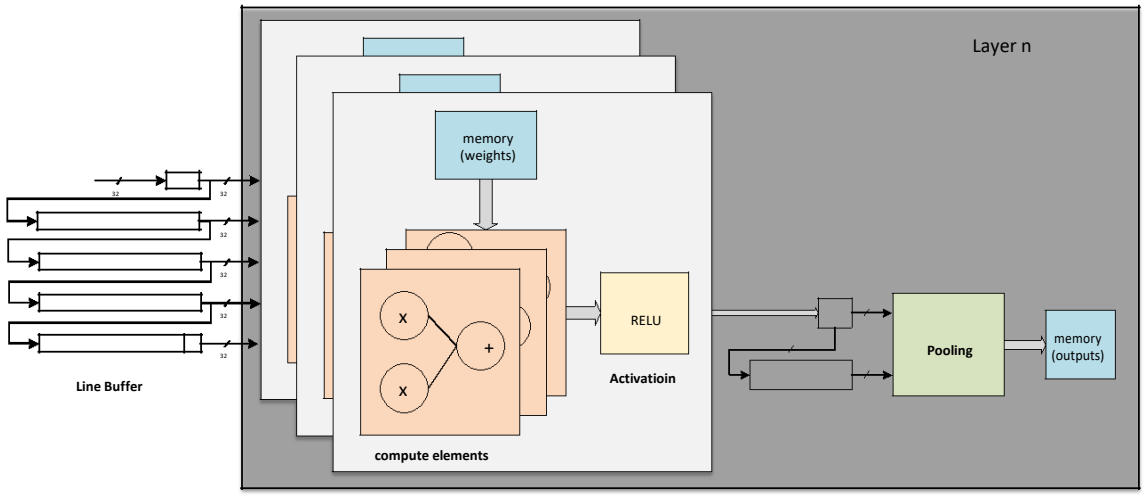

Figure 2. High level architecture of hardware implementation

All four operations are done in parallel with pipelined arithmetic units which computes one operation per clock cycle i.e. for 5x5 convolution, 25 multiplications happen in parallel and 25 additions happen per row wise requiring 3 clocks for each row and another 3 clocks to sum all the sums of rows. Subsequent output will go to activation unit where RELU function is implemented. Number of RELU instantiations are same as number of filters in that layer. Due to line buffer length and single arithmetic operation per cycle, there will be initial latency but once the buffers are full, all units will be active and will work in pipelined fashion. Since CNN operations are monotonous and repetitive, there is no need to stall the pipeline as there are no possible scenarios of pipeline hazards which disrupt the smooth execution of pipeline. Thus the design can operate at higher frequencies. We have pipeline flops at output of every compute operations.

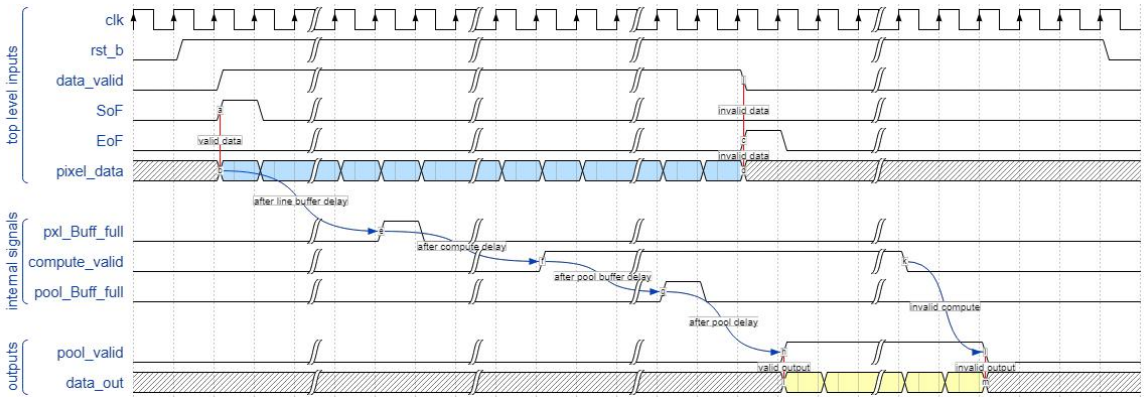

Figure 3. Timing diagram of CNN implementation

Weights and bias are fetched from local memory unit when line buffer is being filled. This will be per filter wise depending on the number of filters present in the layer. Above line buffer and arithmetic operation implementations are repeated for pooling but with a smaller length depending on filter size of pooling. Currently max pool is used for simpler implementation. All outputs are dumped into memory which serves as input for next layer of operation. This completes one layer of operation. Timing diagram of CNN implementation with SOF, pixel data and pooling is shown in figure 3. Compute and pool valid signals tells when the data is valid for compute operations. Buff_full signals tells the start of convolution/poll operation. Output memory is written only when data valid signals are considered for compute.

B. FCL implementation

Same architecture with different sets of parameters can be utilized to realize FCL [4]. The ability to convert FCL operation as CNN operation helps in CNN hardware reuse. Essentially both CNN and FC layers neurons compute dot products. Only difference is the CNN layer neurons are connected to only to part of input and they share the weight filters as the filter strides over the input. This gives us advantage in terms of smaller gate count as same hardware is used for both CNN and FCL. FCL computation flow through the CNN hardware is assisted via the FCL control FSM as show in below figure 4. The FSM designed accommodates the control flow for both loading and reading the weights from SRAM memories. However, the data path implemented in the design is only for reading weights and FCL compute operations. The sram are pre-initialized with the trained weights before starting the computation.

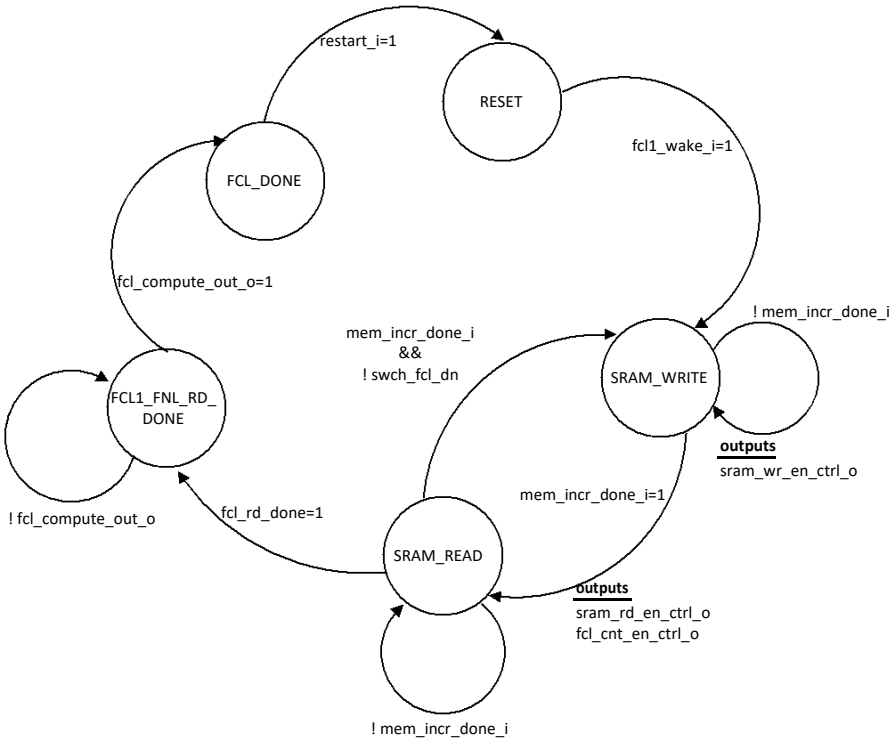

Figure 4. FSM of FCL implementation

Top level signal fcl1_wake_i triggers the FCL computation FSM. In SRAM_WRITE state, trained weights are loaded to sram instances. Once sram write is done, mem_inc_done_i is asserted and this initiates state transition to SRAM_READ. In SRAM_READ state, the weights loaded in previous state are written to weight registers block which route the filter weights to computation blocks. A counter is used to keep a track of number of sets of weights. Transitions between SRAM_READ and SRAM_WRITE keeps on alternating until last set of filter weights are read and swch_fcl_done signal gets asserted indicating last iteration is complete. This initiates a state transition to FCL1_FNL_RD_DONE. The enable to computation block is generated in SRAM_READ state. This is flopped to accommodate the compute delay and serves as data valid indicator for the compute outputs.

C. Scalable RTL parameters

This architecture is made scalable by altering the elaboration time parameters of RTL design. Few of the parameters which can be tweaked to implement different layers of a topology are listed in table 1. The scalability intent of the design helps in implementing FCL operation just by tweaking the parameters of CNN hardware. All other flows are same as CNN. Arithmetic and pooling units in the design currently implements integer arithmetic. Fixed point or INT8 can be swapped for integer unit as an enhancement. For data validation, Lenet software model was used to implement with integer conversion of network parameters. Lenet layers were implemented in the hardware.

Table I. RTL parameters

RTL Parameter	Description
NUM_STRIDE_LEN	No. of parallel convolution operation
NUM_FILTER	No. of filters in the layer
LB_NUM_SHIFT_CONV	Length of line buffer for convolution
LB_NUM_TAPS_CONV	No. of outputs from buffer for convolution
LB_NUM_SHIFT_POOL	Length of line buffer for pooling
LB_NUM_TAPS_POOL	No. of outputs from line buffer for pooling
PIXEL_WIDTH	Width of a pixel before/after scaling

This solution can be utilized in quick inference engines used in Internet of Things (IoT) applications. For example, implementing a detection system to check if there are any human faces in the picture to unlock a gate, automatic food dispensing machines for animals, off/on of appliances based on the activity. Optical character recognition (OCR) to identify numbers and texts etc.

IV. RESULTS

Lenet software model in tensorflow was trained with integer values to obtain weights and bias values with accuracy >90%. This was used in hardware to validate the operation and was found to be matching in RTL simulation environment using VCS. % similarity between software and hardware was 100% as integer operation was used. RTL model was synthesized using design compiler. Table 2 gives few outputs of the synthesis. As the design is heavily pipelined, higher frequencies can be used for operation as we can see that minimum positive slack is 556ps, we have lot of room to increase the frequency further.

Table II. Synthesis report

Parameter	Value
Frequency	400MHz
Gate count (for 1st Lenet layer)	4,74, 373
Total dynamic power	19.527mW
Minimum slack	+556.87ps

To verify hardware results with software, gray scale images of MNIST dataset was converted to hex data with little endian format and pre-loaded to memory units of HW with readmemh utility. This was done with proper conversion and rearrangement of matrix. Once the compute operation was completed, memory contents was dumped to a file and comparison with software outputs was done. All the above steps was done using python numpy package.

A simple scalable hardware for neural network was implemented in hardware with same architecture customizable for both CNN and FCL which can render various application in IoT by interpreting noisy real world data from sensor rich system. With many parallel compute operations, pipelined design and data reuse, high throughput and high frequency applications can be implemented. With modular design, arithmetic blocks can be swapped out for different implementation such as fixed point, INT8, Average pool. This provides a quick and easy way to implement inference engine in hardware. This design was tested for first convolution layers and fully connected layer of Lenet. This design can act like a building block of larger neural networks to implement more complex topology and applications.

A. Future scope

Hardware design was done keeping simplicity in mind. As a result, integer operations was used in each layer and for non-linearity only RELU function was used. As the design is modular with swappable arithmetic units with hand coded modules, floating point arithmetic units can be instantiated to accommodate higher accuracy operations.

With floating point arithmetic units, it will also help if nonlinear functions other than RELU is also implemented such as tanh, sigmoid etc. More optimizations such as gate count reductions to decrease power and area with the trade-off of frequency can be carried out to fit the design in smaller applications.

REFERENCES

[1] Farabet, Clément, et al. "Cnp: An fpga-based processor for convolutional networks." 2009 International Conference on Field Programmable Logic and Applications. IEEE, 2009.

[2] DiCecco, Roberto, et al. "Caffeinated FPGAs: FPGA framework for convolutional neural networks." 2016 International Conference on Field-Programmable Technology (FPT). IEEE, 2016.

[3] Abadi, Martín, et al. "Tensorflow: Large-scale machine learning on heterogeneous distributed systems." arXiv preprint arXiv:1603.04467 (2016).

[4] Cs231n.github.io. (2019). CS231n Convolutional Neural Networks for Visual Recognition. [online] Available at: http://cs231n.github.io/convolutional-networks/.