ABSTRACT

Most neural network designs for FPGAs are inflexible. In this paper, we propose a flexible VHDL structure that would allow any neural network to be implemented on multiple FPGAs. Moreover, the VHDL structure allows for testing as well as training multiple neural networks. The VHDL design consists of multiple processor groups. There are two types of processor groups: Mini Vector Machine Processor Group and Activation Processor Group. Each processor group consists of individual Mini Vector Machines and Activation Processor. The Mini Vector Machines apply vector operations to the data, while the Activation Processors apply activation functions to the data. A ring buffer was implemented to connect the various processor groups.

Keywords  FPGAs, Neural Networks, Codesign, Microcode

1 Introduction

Neural networks excel at a wide variety of tasks. Tasks such as speech recognition, noise filtering, and text prediction are easily solved using neural networks. However, there are many downsides to neural networks. Neural networks require large amounts of data to train and test on. Moreover, neural networks require very powerful processors to compute the matrix operations. As result, the processors’ computational power limits the speed of the neural networks.

CPUs are an obvious choice to train and test neural networks. CPUs are general purpose processors that can handle any task. In spite of the CPUs’ flexibility, CPUs are very inefficient at computing neural networks as CPUs are not optimized for matrix operations. On the other hand, co-processors such as Nvidia Tesla [1] and Intel Xeon Phi [2] are better at processing neural networks when compared to CPUs. The co-processors employ a large array of specialized processors. The large array of processors enables the co-processors to massively speed up the matrix operations. Despite co-processors being very powerful, they have memory bandwidth limitations. Most co-processors use PCIe 3.0 x16 to retrieve data, which is limited to 16 GB/s. The memory bandwidth also bottlenecks the neural networks’ computations. Furthermore, co-processors require a CPU for control, which adds cost and latency.

FPGAs are a possible solution to the problems presented above. FPGAs have better memory bandwidth/cost ratios when compared to co-processors. Moreover, FPGAs do not require a CPU for control. The FPGAs’ flexibility allows the FPGAs to adapt to different types of neural networks. Therefore, FPGAs are a cost efficient solution for processing neural networks. The literature contains numerous examples of FPGA frameworks optimized towards neural networks. The paper, "SpWA: an efficient sparse winograd convolutional neural networks accelerator on FPGAs" [3], shows a CNN implemented in Vivado HLS. Another paper, "Runtime Programmable and Memory Bandwidth Optimized FPGA-Based Coprocessor for Deep Convolutional Neural Network" [4], proposes a re-programmable DCNN accelerator using FSM based processors. The paper [4] also uses advance caching to minimize the load times of the data. A similar paper, "Hardware/Software Codesign for Convolutional Neural Networks Exploiting Dynamic Partial Reconfiguration on PYNQ" [5], shows the codesign of a CNN on the Xilinx ZYNQ. In the paper [5], the ARM cores load data from the RAMs, while the FPGA executes the CNN’s matrix operations.
This paper proposes an FPGA solution to the problems above. The solution consists of an assembler and a VHDL design. The assembler takes in neural network assembly codes and produces microcodes. The microcodes are flashed onto a cluster of FPGA. The cluster of FPGAs executes multiple neural networks in parallel, which accelerates the training and testing phases. Furthermore, the cluster of FPGAs allows for a greater memory bandwidth. The cluster of FPGAs overcomes the memory bandwidth limitations of individual FPGAs.

1.1 Multi-Layer Perceptions

Let $X_i$ = data input vector of layer $i$
Let $W_i$ = weight matrix of the layer $i$
Let $B_i$ = bias vector of the layer $i$
Let $A(V)$ = activation function with respect to matrix $V$
Let $O_i$ = layer output vector of layer $i$

$$O_i = A(W_i^T X_i + B_i)$$  \hspace{1cm} (1)

ReLU($x$) = $\max(0, x)$ \hspace{1cm} (2)

Multi-layer perceptions (MLPs) \cite{6} are a type of neural network. MLPs have an input layer, multiple hidden layers, and an output layer. The input data enters a layer through the input vector $X_i$. Then the input vector $X_i$ is multiplied by the weights $W_i^T$. After matrix multiplication, biases $B_i$ are added to the $W_i^T X_i$. After that, the result passes through the activation function $A(V)$ and produces the layer’s output $O_i$. There are many types of activation functions used in neural networks. For example, Eqn. 2 shows the ReLU activation function. The ReLU activation function sets all negative numbers to zero. Overall, the input data goes through many layers until the final result is produced at the output layer.

2 Design Overview and Requirements

The goal of the project is to accelerate multiple neural networks using multiple FPGAs. The targeted FPGA boards must use Xilinx’s 7 Series FPGAs. All the FPGA boards must be identical. Moreover, the FPGA boards must have onboard flash, RAM, and system buses. Fig. 1 shows the neural network processor and assembler. The Matrix Assembler is a high level optimizing assembler, which parses the neural network assembly codes. The Matrix Assembler parses as many neural network assembly codes as the user wants. After parsing the assembly codes, the Matrix Assembler optimizes the assembly codes and neural network processors. Then the Matrix Assembler generates the VHDL codes and the microcodes. The VHDL codes contain the structure of the Matrix Machine. The Matrix Machine consists of multiple Mini Vector Machines. Each Mini Vector Machine computes a vector operation using a single DSP. The DSPs are set to process 16 bit signed integers. 16 bit precision is enough for most of the neural network applications. When the Mini Vector Machines are put together, the Mini Vector Machines perform matrix operations. The Mini Vector Machines allow the Matrix Machine to adapt to different sizes of matrices. Subsequently, the VHDL codes are synthesized into the bit-streams using the Xilinx’s Vivado Design Suite. After generating the bit-streams, the bit-streams are flashed to the onboard flash. The onboard flash then loads the bit-stream onto the FPGA. The system buses transfer the neural network data and microcode from the control server to the onboard RAM. The onboard RAM acts as a buffer for the FPGA. The microcodes schedule the execution of the Matrix Machine by coordinating the individual Mini Vector Machines.

For the functional requirements, the Matrix Machine must train and test MLPs. The Matrix Machine must calculate the forward passes of the MLPs. After calculating the forward passes, the loss functions’ gradients must be calculated using the back-propagation algorithm. The gradients are then used to update the weights of the MLPs. In order to be flexible, the VHDL design must be generalized to run any type of MLP. Firstly, the Matrix Assembler must handle any number of MLPs regardless of the number of FPGAs. Secondly, the Matrix Machine must handle matrices of any size and shape. The input matrices, the weight matrices, and the bias matrices could be as big as the user wants. Thirdly, the Matrix Machine must be able to dynamically load different MLPs at runtime. In other terms, the Matrix Machine must be able to switch between different MLPs without regenerating the bit-stream. Fourthly, the Matrix Machine must scale to any number of LUTs, BRAMs, and DSPs. If the Matrix Assembler detects the FPGA has a high number of DSPs, then the Matrix Assembler generates more Mini Vector Machines to take advantage of the DSPs. If the Matrix Assembler detects the FPGA has a low number of DSPs, then the Matrix Assembler reduces the number of Mini Vector Machines. Lastly, the Matrix Machine must scale to any number of FPGAs. If the number of MLPs is greater than the
number of FPGAs, then the MLPs are processed sequentially. If the number of MLPs is less than the number of FPGAs, then the MLPs are divided and are processed in parallel. If the number of MLPs is equal the number of FPGAs, then the Matrix Assembler maps 1 MLP to 1 FPGA.

3 Matrix Assembler: High Level Optimizing Assembler

The Matrix Assembler takes in neural network assembly codes and produces instructions and VHDL codes. At runtime, the instructions are decoded into microcodes. The decoding is done to reduce the size of the instruction cache. Moreover, the Matrix Assembler controls the number of processor groups and the types of processors using the VHDL codes. As a result, the Matrix Assembler is able to optimize the VHDL codes for a specific FPGA.
Table 1: Neural network assembly codes.

| Assembly | ARG0 | ARG1 | ARG2 | ARG3 | ARG4 | Description                              |
|----------|------|------|------|------|------|------------------------------------------|
| INPUT    | OUTMAT | SIZEN | SIZEM | NONE | NONE | Loads an N X M data matrix               |
| WEIGHT   | OUTMAT | SIZEN | SIZEM | NONE | NONE | Loads an N X M weight matrix             |
| BIAS     | OUTVEC | SIZEN | NONE  | NONE | NONE | Loads a bias vector with size N          |
| ACT      | OUTVEC | SIZEN | NONE  | NONE | NONE | Loads an activation lookup table with size N |
| MLP      | OUTMAT | INMAT | INMAT | INVEC | INVEC | Executes a MLP layer                     |
| OUTPUT   | INMAT | NONE  | NONE  | NONE | NONE | Stores data matrix                       |

3.1 Assembly Codes

Table 1 shows the neural network assembly codes. INPUT code specifies the input matrix to the neural network. WEIGHT, BIAS, ACT, and MLP codes define the structure of a single layer. The OUTPUT code controls the output matrix of the neural network.

3.2 Instruction Set Architecture

| Instruction               | Op code | Description                                      |
|---------------------------|---------|-------------------------------------------------|
| VECTOR_DOT_PRODUCT        | 000     | Vector dot product                              |
| VECTOR_SUMMATION          | 001     | Vector summation                                |
| VECTOR_ADDITION           | 010     | Vector addition                                 |
| VECTOR_SUBTRACTION        | 011     | Vector subtraction                              |
| ELEMENT_MULTIPLICATION     | 100     | Element wise multiplication                      |
| ACTIVATION_FUNCTION       | 101     | Apply activation function to vectors            |
| NOP                       | 110     | No operation                                    |

Table 2: Instruction set architecture.

The Matrix Assembler translates the assembly codes to the instructions. Table 2 shows the list of instructions. Matrix multiplication is achieved by using multiple vector dot operations. Moreover, matrix addition is achieved using by multiple vector additions. Fig. 2 shows the bit arrangement for the instruction architecture. The operation code controls the type of operation, while the number of iterations controls the number of loops. Moreover, the operation code is applied to the processors designated by the processor select start and the processor select end. For the 32 bit version, the instructions only control a maximum of 128 processor groups. For the 48 bit version, the instructions only control a maximum of 1024 processor groups.

3.3 Microcode

The Matrix Assembler also translates the instructions to microcode. Fig. 3 shows the 32 bit microcode. Each microcode controls 4 MVMs. The MVMs are arranged in groups of 4 because the 4:1 multiplexer is the most efficient multiplexer.
Figure 3: Microcode bit arrangement.

The 4:1 multiplexer uses the least amount of LUTs and has the lowest latency. microcode(9..0) controls the number of cycles in a microcode. The number of cycles allows the Matrix Assembler to execute a given microcode for any length of time. microcode(10) controls the selection of the input columns. If input column 0 is selected, then the input data is written to column 0. If input column 1 is selected, then the input data is written to column 1. microcode(11) controls the activation of the input counter. If the input counter is enabled, then the input counter increments at every cycle. The input counter’s value is feed into the input addresses of the individual MVMs. microcode(12) controls the selection of the output columns. microcode(13) controls the activation of the output counter. microcode(15..14) controls the selection of the output 4:1 multiplexer. The output 4:1 multiplexer controls the output of the processor group. microcode(31..16) contains 4 processor control signals. Each processor control signal is mapped to a MVM input processor control signal.

3.4 Resource Allocation

| Component   | LUTs | FFs  | RAMB18Ks | DSPs |
|-------------|------|------|----------|------|
| MVM_PG      | 495  | 1642 | 8        | 4    |
| ACTPRO_PG   | 447  | 1406 | 12       | 0    |

Table 3: Processor group resource usages.

Let $N_{DDR}$ = number of 32 bit DDR RAM channels
Let $CLK_{DDR}$ = DDR RAM clock in MHz
Let $CLK_{FPGA}$ = FPGA clock in MHz
Let $LUT_{FPGA}$ = number of leftover DSPs on the FPGA
Let $LUT_{ACTPRO\_PG}$ = number of DSPs used by the ACTPRO\_PG
Let $FF_{FPGA}$ = number of leftover FFs on the FPGA
Let $FF_{ACTPRO\_PG}$ = number of FFs used by the ACTPRO\_PG
Let $BRAM_{FPGA}$ = number of leftover block RAMs on the FPGA
Let $BRAM_{ACTPRO\_PG}$ = number of block RAMs used by the ACTPRO\_PG
Let $N_{MVM\_pg}$ = optimal number of Mini Vector Machine processor groups
Let $N_{ACTPRO\_PG}$ = optimal number of Activation processor groups
\[
N_{\text{MV M\_PG}} = \frac{N_{\text{DDR}} \cdot \text{CLK}_{\text{DDR}}}{\text{CLK}_{\text{FPGA}}} \quad (3)
\]

\[
N_{\text{ACTPRO\_PG}} = \min \left( \frac{\text{LUT}_{\text{FPGA}}}{\text{LUT}_{\text{ACTPRO\_PG}}}, \frac{\text{FF}_{\text{FPGA}}}{\text{FF}_{\text{ACTPRO\_PG}}}, \frac{\text{BRAM}_{\text{FPGA}}}{\text{BRAM}_{\text{ACTPRO\_PG}}} \right) \quad (4)
\]

The Matrix Assembler determines the optimal number of processor groups in order to fully utilize the FPGA’s resources. Eqn. 3 shows the equation for the optimal number of Mini Vector Machine processor groups \(N_{\text{MV M\_PG}}\). The number of Mini Vector Machine processor groups \(N_{\text{MV M\_PG}}\) is only limited by the number of DDR RAM channels \(N_{\text{DDR}}\). Furthermore, Table 3 shows the resource usages of each processor group. The optimal number of Activation processor groups \(N_{\text{ACTPRO\_PG}}\) is calculated using Table 3 and Eqn. 4.

4 Matrix Machine: Neural Network Processors

Fig. 4 shows the Matrix Machine. The Matrix Machine contains a global controller that coordinates multiple processor groups. The global controller first decodes the instructions into microcodes. Then the global controller writes microcodes and data to a circular FIFO. The FIFO’s purpose is to distribute the microcodes and data to each processor group. The FIFO also collects outputs of each processor group. Moreover, the FIFO reduces the propagation delay of the signals. Each processor group has a local controller, which receives microcodes from the global controller. The local controller’s purpose is to cache the microcodes. The microcode cache reduces the number of load operations and minimizes the propagation delay. Moreover, each processor group consists of Mini Vector Machines (MVMs) or Activation Processors (ACTPROs). The Mini Vector Machines execute vector operations, while the Activation Processors execute activation functions.
### 4.1 Processor Groups

| Signal                  | Direction | Description                                      |
|-------------------------|-----------|--------------------------------------------------|
| CLK                     | IN        | Clock                                            |
| group_control(1..0)     | IN        | Control signal for execution                     |
| microcode(31..0)        | IN        | Microcode input                                  |
| input_data0(15..0)      | IN        | Input data port 0                                |
| input_data1(15..0)      | IN        | Input data port 1                                |
| output_data0(15..0)     | OUT       | Output data port 0                               |
| output_data1(15..0)     | OUT       | Output data port 1                               |

Table 4: Mini Vector Machine processor group ports.

![Figure 5: MVM processor group.](image)

Table 4 shows the ports of the MVM processor group. The group control port starts and stops the executions of the processor group. Furthermore, the microcode input port is written to the microcode cache. The microcode cache is used to minimize the load penalties. After writing the microcodes, the microcodes are decoded and are sent to the individual processors. The microcode controls the counters, number of cycles, and the type of operation. Each processor group has two input data ports and two output data ports. Each input port receives a 16 bit integer. The output ports transmit 16 bit integers.

The structure of the MVM processor group is presented in Fig. 5. The MVM processor group consists of 4 processors joined together by 1 x 4:1 multiplexer, 1 x microcode cache, and 1 x local controller. The processors are arranged in groups of 4 because the 4:1 multiplexer is the most efficient multiplexer. Each MVM processor group uses 495 LUTs, 1642 FFs, 4 x DSP48E1, and 8 x RAMB18Ks in total. The microcode cache stores 16 microcodes in total. The 8 bit input counter is used to select the input addresses of the MVMs. The input counter allows the MVMs to load the vectors column-wise. Column-wise vector loading enables the MVMs to cache the column vectors in order to minimize the load penalties. The 8 bit output counter is used to store vectors column-wise. The output counters are designed to mirror the input counters. The output multiplexer is used to select the outputs of the MVMs.

Let $T_{cycle} = \text{period of a cycle in seconds}$
Let $N_{bits} = \text{number of bits per element}$
Let $N_e = \text{number of elements per processor}$
Let $N_{proc} = \text{number of processors per group}$
Let $N_I = \text{number of iterations}$
Let $C_{STALL} = \text{number of stall cycles per iteration}$
Let $C_{LOAD} = \text{number of load cycles per iteration}$
Let $C_{RUN}$ = number of run cycles per iteration
Let $C_{STORE}$ = number of store cycles per iteration
Let $T_{RUN}(N_I)$ = total number of run cycles for a given number of iterations $N_I$
Let $T_{all}(N_I)$ = total number of cycles for a given number of iterations $N_I$
Let $E(N_I)$ = efficiency for a given number of iterations $N_I$
Let $P(N_I)$ = processing rate in $\frac{\text{elements}}{s}$ for a given number of iterations $N_I$
Let $R(N_I)$ = data throughput in Mb/s for a given number of iterations

$$T_{RUN}(N_I) = N_{proc} \cdot N_I \cdot C_{RUN}$$ (5)

$$T_{all}(N_I) = N_{proc} \cdot ((N_I + N_{proc}^2 - 1) \cdot (C_{LOAD}) + N_I \cdot (C_{RUN} + C_{STORE} + C_{STALL}))$$ (6)

$$E(N_I) = \frac{T_{RUN}(N_I)}{T_{all}(N_I)}$$ (7)

$$P(N_I) = \frac{N_{proc}^2 \cdot N_I \cdot N_e}{T_{all}(N_I) \cdot T_{cycle}}$$ (8)

$$R(N_I) = P(N_I) \cdot N_{bits} \cdot 1 \times 10^{-6}$$ (9)

For vector addition and $N_I = 1024$ iterations, the total number of run cycles and total number of cycles are calculated below. Also the efficiency and processing rate are calculated.

$T_{RUN}(1024) = 4 \cdot 1024 \cdot 519 = 2125824$

$T_{all}(1024) = 4 \cdot ((1024 + 4^2 - 1) \cdot (256) + (1024) \cdot (519 + 256 + 0)) = 4238336$

$E(1024) = \frac{T_{RUN}(1024)}{T_{all}(1024)} = \frac{2125824}{4238336} = 0.501$

$P(1024) = \frac{4^2 \cdot 1024 \cdot 1024}{4238336 \cdot 10 \times 10^{-6}} = 3.95 \times 10^8 \frac{\text{elements}}{s}$

$R(1024) = 3.95 \times 10^8 \frac{\text{elements}}{s} \cdot 16 \text{bits} \cdot 1 \times 10^{-6} = 6320 \frac{\text{Mb}}{s}$

For vector dot product and $N_I = 1024$ iterations, the total number of run cycles and total number of cycles are calculated below. Also the efficiency and processing rate are calculated.

$T_{RUN}(1024) = 4 \cdot 1024 \cdot 519 = 2125824$

$T_{all}(1024) = 4 \cdot ((1024 + 4^2 - 1) \cdot (256) + (1024) \cdot (519 + 0 + 248) + 256) = 4206592$

$E(1024) = \frac{T_{RUN}(1024)}{T_{all}(1024)} = \frac{2125824}{4206592} = 0.505$

$P(1024) = \frac{4^2 \cdot 1024 \cdot 1024}{4206592 \cdot 10 \times 10^{-6}} = 3.99 \times 10^8 \frac{\text{elements}}{s}$

$R(1024) = 3.99 \times 10^8 \frac{\text{elements}}{s} \cdot 16 \text{bits} \cdot 1 \times 10^{-6} = 6384 \frac{\text{Mb}}{s}$

For the activation function and $N_I = 1024$ iterations, the total number of run cycles and total number of cycles are calculated below. Also the efficiency and processing rate are calculated.

$T_{RUN}(1024) = 4 \cdot 1024 \cdot 517 = 2117632$

$T_{all}(1024) = 4 \cdot ((1024 + 4) \cdot (512) + (1024) \cdot (517 + 256 + 0)) = 5271552$

$E(1024) = \frac{T_{RUN}(1024)}{T_{all}(1024)} = \frac{2117632}{5271552} = 0.401$

$P(1024) = \frac{4^2 \cdot 1024 \cdot 1024}{5271552 \cdot 10 \times 10^{-6}} = 3.18 \times 10^8 \frac{\text{elements}}{s}$

$R(1024) = 3.18 \times 10^8 \frac{\text{elements}}{s} \cdot 16 \text{bits} \cdot 1 \times 10^{-6} = 5088 \frac{\text{Mb}}{s}$
The processor groups have high efficiency as the efficiency approaches 50% for vector operations. Moreover, each processor group processes elements at a rate of $> 5000 \frac{Mb}{s}$, which is $\frac{1}{5}$ the bandwidth of a 32 bit DDR2 RAM.

### 4.2 Mini Vector Machines

| Signal          | Direction | Description                        |
|-----------------|-----------|------------------------------------|
| CLK             | IN        | Clock                              |
| processor_control(2..0) | IN        | Operation code                     |
| processor_control(3) | IN        | Right BRAM MSB select              |
| input_data0(15..0)    | IN        | Input data port 0                  |
| input_addr0(15..0)     | IN        | Input address port 0               |
| input_data1(15..0)    | IN        | Input data port 1                  |
| input_addr1(15..0)    | IN        | Input address port 1               |
| output_data0(15..0)   | OUT       | Output data port 0                 |
| output_addr0(15..0)   | OUT       | Output address port 0              |
| output_data1(15..0)   | OUT       | Output data port 1                 |
| output_addr1(15..0)   | OUT       | Output address port 1              |

Table 5: Mini Vector Machine ports.

| processor_control(2..0) | Operation name | Operation description          |
|-------------------------|----------------|---------------------------------|
| 000                     | MVM_RESET      | Reset all registers             |
| 001                     | MVM_READ       | BRAM read                       |
| 010                     | MVM_WRITE      | BRAM write                      |
| 011                     | MVM_VEC_DOT    | Vector dot product using BRAM   |
| 100                     | MVM_VEC_SUM    | Vector summation using BRAM     |
| 101                     | MVM_VEC_ADD    | Vector addition using BRAM      |
| 110                     | MVM_VEC_SUB    | Vector subtraction using BRAM    |
| 111                     | MVM_ELEM_MULTI | Element wise multiplication     |

Table 6: Mini Vector Machine processor control.

The Mini Vector Machine’s purpose is to execute vector operations. Tab. [5] shows the Mini Vector Machine’s ports. The Mini Vector Machine uses clocks of 100MHz, 100MHz, 300MHz, and 500MHz for Spartan-7, Artix-7, Kintex-7, and Virtex-7 respectively. The processor control signal is shown in Tab. [6]. The processor control signal allows the Mini Vector Machine to run vector dot product, vector summation, vector addition, and vector subtraction. Moreover, the processor control signal manages the BRAMs’ reading and writing. The Mini Vector Machine has 2 input ports and 1 output port. The input ports have input data lines and input address lines. The input ports allow vectors to be written to...
the left BRAM. The output port has a output data line and a output address line. The output port allows vectors to be read from the right BRAM.

Fig. 6 shows the structure of the Mini Vector Machine. The Mini Vector Machine consists of 1 x DSP48E1, 2 x BRAM, 2 x counter, and control logic. The control logic requires 50 LUTs and 210 FFs. Each BRAM (RAMB18E1) stores 1024 x 16 bit signed value. Furthermore, each BRAM has two read/write ports. The left BRAM’s dual outputs are feed to the dual inputs of the DSP48E1. Then the DSP48E1 performs arithmetic on the DSP48E1’s inputs. After computing the values, the DSP48E1 outputs a 48 bit signed result. Subsequently, the 48 bit signed integer is truncated into a 16 bit signed integer. The DSP48E1’s single output is connected to the right BRAM’s port 0. The right BRAM’s port 0 is always set to write DSP48E1’s output, while port 1 is always set to read the right BRAM’s data.

Fig. 7 shows the writing timing diagram of the Mini Vector Machine. Mini Vector Machine starts with the MVM_READ state, where Mini Vector Machine is halted. Then the Mini Vector Machine’s state transitions to MVM_WRITE. In the MVM_WRITE state, the Mini Vector Machine executes the setup phase of the left BRAM in the 1st cycle. In the 2nd cycle, the left BRAM writes input_data0 and input_data1 in parallel using the addresses given by input_addr0 and input_addr1. Input_data0 and input_data1 each have a 16 bit signed integer. Moreover, the left BRAM takes 1 cycle to write the input data pairs into the columns.

Once the left BRAM is full, the Mini Vector Machine executes the vector operations. Fig. 8 shows the Mini Vector Machine’s vector addition. The 1st cycle is used for the setup phase of the DSP48E1, BRAMs, read counter, and write counter. In the 2nd cycle, the left BRAM is read using the read counter. At the same time, the read counter is incremented. In the 3rd cycle, the DSP48E1’s A and B ports are feed with the left BRAM’s data. The DSP48E1 is configured as a 6 stage pipeline. At the 8th cycle, the DSP48E1’s P port outputs the result. Also in the 8th cycle, the write counter increments. In the 9th cycle, the right BRAM writes the result using the write counter.

4.3 Activation Processors

The Activation Processor performs bit shifts and executes the activation function. The Activation Processor’s ports are similar to the Mini Vector Machine’s ports shown in Table 5. The only difference is the size of the processor control signal. Table 7 shows the list of controls for the Activation Processor.
| processor_control(1..0) | Operation name       | Operation description                        |
|------------------------|----------------------|----------------------------------------------|
| 00                     | ACTPRO_READ          | Read BRAM                                    |
| 01                     | ACTPRO_WRITE_ACT     | Write activation function to BRAM            |
| 10                     | ACTPRO_WRITE_DATA    | Write input data to BRAM                     |
| 11                     | ACTPRO_RUN           | Bit shift and activation function            |

Table 7: Activation Processor operations.

Fig. 9 shows the structure of the Activation Processor. Activation Processor consists of 3 x BRAM, 2 x counter, and 1 x control logic. The control logic requires 70 LUTs and 210 FFs. The left BRAM is connected to the dual bit shifts. Each bit shifter applies a 7 bit shift to the right. After the dual bit shifts, the values are used as addresses to look-up the results for the activation functions. Each look-up table uses 1 BRAM resource. Moreover, the look-up tables are able to store the activation functions as well as the derivatives of the activation functions. At the end, the results are written to the right BRAM.

Fig. 10 shows the Activation Processor executing the ReLU function. In the 1st cycle of ACTPRO_RUN, the control logic sets up the pipeline. At the 2nd cycle, the control logic reads the left BRAM using the read counter. At the same time, the read counter is incremented. In the 3rd cycle, the Activation Processor shifts the 2 x 16 bit integer. In the 5th cycle, result of the activation function is retrieved. In the 6th cycle, the write counter is incremented. In the 7th cycle, the result is written to the right BRAM using the write counter.

5 Performance/Cost Evaluation

Let $N_{DDR} =$ number of DDR RAM channels
Table 8: Performance/Cost evaluation of FPGAs.

| FPGA          | IO pins | DDR channels | DDR Bus Clock (MHz) | Cost (CAD) | DDR/Cost (Mb/s/CAD) |
|---------------|---------|--------------|---------------------|------------|---------------------|
| XC7S50-1      | 250     | 2            | 333.33              | 75.94      | 561.84              |
| XC7S75-1      | 400     | 4            | 333.33              | 134.46     | 634.63              |
| XC7S100-1     | 400     | 4            | 333.33              | 163.73     | 521.17              |
| XC7S50-2      | 250     | 2            | 400                 | 95.11      | 538.32              |
| XC7S75-2      | 400     | 4            | 400                 | 147.95     | 692.12              |
| XC7S100-2     | 400     | 4            | 400                 | 198.12     | 516.85              |
| XC7A75T-1     | 300     | 3            | 333.33              | 213.27     | 300.08              |
| XC7A100T-1    | 300     | 3            | 333.33              | 234.6      | 272.80              |
| XC7A200T-1    | 500     | 5            | 333.33              | 381.95     | 279.26              |

Let $CLK_{DDR}$ = DDR bus clock in MHz

Let $N_{bits}$ = number of bits on the DDR RAM bus

Let $C_{FPGA}$ = the cost of the FPGA in CAD

Let $R = DDR$ throughput in $\frac{Mb}{s}$

Let $F = DDR$ throughput to cost ratio in $\frac{Mb}{s/CAD}$

$$R = CLK_{DDR} \cdot 2 \cdot N_{bits} \cdot N_{DDR}$$ (10)

$$F = \frac{R}{C_{FPGA}}$$ (11)

The main limiting factor in the FPGAs’ performances is the DDR throughput $R$. Table 8 [10] [11] [12] shows the performance/cost evaluation of FPGAs. Only the Spartan-7 and Artix-7 families were considered because they have the highest performance/cost ratio. Firstly, the FPGAs’ DDR throughputs $R$ were calculated using the Eqn. [10]. Secondly, the performance/cost ratios $F$ were calculated using the costs of the FPGAs and Eqn. [11]. Finally, Spartan-7 XC7S75-2 was selected as the best FPGA because the XC7S75-2 has the highest performance/cost ratio. Moreover, a cluster of FPGAs could built using the XC7S75-2. The cluster would outperform a standalone FPGA because the cluster has a higher number of DDR channels.

6 Conclusion

Neural networks prove to be extremely useful. However, neural networks require a lot of computational power. Moreover, neural networks need a large memory bandwidth to load the data. FPGAs were selected to solve the problems because FPGAs have a high memory bandwidth/cost ratio. Spartan-7 XC7S75-2 was selected because XC7S75-2 has the best bandwidth/cost ratio out of the Xilinx’s 7 series FPGAs. Moreover, the Matrix Assembler was implemented to optimize the design of the Matrix Machine. The Matrix Assembler takes in neural network assembly codes and produces microcodes and VHDL codes. The VHDL codes form the structure of the Matrix Machine. The Machine Machine has multiple Mini Vector Machines that execute vector operations. The Mini Vector Machines allow neural network acceleration. Furthermore, the microcodes were used to schedule the executions of the Mini Vector Machines. The microcodes allow the FPGAs to switch neural networks without reloading the bitstream.

References

[1] S. Chetlur, C. Woolley, P. Vandermersch, J. Cohen, J. Tran, B. Catanzaro, and E. Shelhamer, “cudnn: Efficient primitives for deep learning,” *arXiv preprint arXiv:1410.0759*, 2014.

[2] L. Jin, Z. Wang, R. Gu, C. Yuan, and Y. Huang, “Training large scale deep neural networks on the Intel Xeon Phi many-core coprocessor,” in *2014 IEEE International Parallel & Distributed Processing Symposium Workshops (IPDPSW)*, pp. 1622–1630, IEEE, 2014.
[3] L. Lu and Y. Liang, “SpWA: an efficient sparse winograd convolutional neural networks accelerator on FPGAs,” in *Proceedings of the 55th Annual Design Automation Conference*, p. 135, ACM, 2018.

[4] N. Shah, P. Chaudhari, and K. Varghese, “Runtime Programmable and Memory Bandwidth Optimized FPGA-Based Coprocessor for Deep Convolutional Neural Network,” *IEEE Transactions on Neural Networks and Learning Systems*, no. 99, pp. 1–13, 2018.

[5] F. Kästner, B. Janßen, F. Kautz, M. Hübner, and G. Corradi, “Hardware/Software Codesign for Convolutional Neural Networks Exploiting Dynamic Partial Reconfiguration on PYNQ,” in *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pp. 154–161, IEEE, 2018.

[6] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, “Learning representations by back-propagating errors,” *nature*, vol. 323, no. 6088, p. 533, 1986.

[7] Xilinx, “UG473: 7 Series FPGAs Memory Resources.” [https://www.xilinx.com/support/documentation/user_guides/ug473_7Series_Memory_Resources.pdf](https://www.xilinx.com/support/documentation/user_guides/ug473_7Series_Memory_Resources.pdf), 2016.

[8] Xilinx, “PG058: Block Memory Generator v8.3.” [https://www.xilinx.com/support/documentation/ip_documentation/blk_mem_gen/v8_3/pg058-blk-mem-gen.pdf](https://www.xilinx.com/support/documentation/ip_documentation/blk_mem_gen/v8_3/pg058-blk-mem-gen.pdf), 2017.

[9] Xilinx, “UG479: 7 Series FPGAs DSP48E1.” [https://www.xilinx.com/support/documentation/user_guides/ug479_7Series_DSP48E1.pdf](https://www.xilinx.com/support/documentation/user_guides/ug479_7Series_DSP48E1.pdf), 2018.

[10] Xilinx, “7 Series FPGAs Data Sheet: Overview.” [https://www.xilinx.com/support/documentation/data_sheets/ds180_7Series_Overview.pdf](https://www.xilinx.com/support/documentation/data_sheets/ds180_7Series_Overview.pdf), 2018.

[11] Xilinx, “Spartan-7 FPGAs Data Sheet: DC and AC Switching Characteristics.” [https://www.xilinx.com/support/documentation/data_sheets/ds189_spartan-7-data-sheet.pdf](https://www.xilinx.com/support/documentation/data_sheets/ds189_spartan-7-data-sheet.pdf), 2018.

[12] Xilinx, “Artix-7 FPGAs Data Sheet: DC and AC Switching Characteristics.” [https://www.xilinx.com/support/documentation/data_sheets/ds181_Artix_7_Data_Sheet.pdf](https://www.xilinx.com/support/documentation/data_sheets/ds181_Artix_7_Data_Sheet.pdf), 2018.