An FPGA Resource Adaptable General Neural Network Accelerator

Chengsen Dong [1], Zheng Xie [2]

School of Engineering
University of Central Lancashire
Preston, UK

[1] cdong1@uclan.ac.uk; [2] ORCID: 0000-0001-8649-6235

***Abstract* - As Artificial Intelligence is becoming embedded in people's lives, the evolution of Internet of Things is moving towards edge computing where the speed and power consumption in data processing is critical. The feature of re-programmability and power efficiency has made FPGA a promising edge processing hardware platform for accelerating deep neural networks. An FPGA 'resource-adaptable' neural network accelerator is proposed in this paper. The architecture and behavior of this accelerator is determined only by the way its C program is designed. The design of the architecture, programmed in C code, is converted to a description in the form of a hardware description language such as VHDL or Verilog. The conversion is carried out by High-Level Synthesis (HLS) software provided by the Xilinx Vivado development package. Since the accelerator architecture is fully parameterized in the C code, it can be tailored freely according to the availability of FPGA logic elements, and hence implemented by different types of FPGA. The proposed accelerator has a configurable register unit, which enables it to dynamically adjust the computing behavior according to the computing requirements of different neural networks without changing the design of architecture.**

Keywords - FPGA, Adaptable Neural Network Accelerator, High level synthesize, Parameterized Architecture, Xilinx.

I. INTRODUCTION

The advent of the Internet of Things [1] has facilitated the gathering of data from devices existing anywhere. It is not enough to simply receive the data. The value of the data lies in the insights that may be drawn from it and how the data is used. Edge-computing performs data analysis in a hybrid of local 'on-premise' processing and 'cloud' processing. Much of the computing is done close to the device which provides great speed, accuracy and reliability. Other computing is done using more powerful facilities further away to achieve a wider range of functionality. A high speed of data processing and low power consumption is critical for a practical edge computing system [2], especially with applications that employ neural networks.

There have been many attempts to accelerate artificial intelligence algorithms in edge-computing systems. In the research of Bogoslovskii et al. [3], an 'approximator' based on a multilayer perceptron and a wavelet neural network is implemented using STM32 microcontrollers, where ARM 'Cortex M series CPUs' play the role of implementing the neural network. Even though ARM Cortex M microcontrollers are low power devices, the overall power consumption involved is still quite high in this implementation. In addition to using CPUs for neural network calculations, there is also a lot of published research on using GPUs to implement and accelerate neural network algorithms. For example, in the research of Jose et al. [4], they used NVIDIA JETSON TX2 to implement a face recognition monitoring system. This system achieves 97% accuracy and 7.5 watts of power consumption. But for some small devices, the power consumption of 7.5 watts is still considered very high. Nevertheless, the internal structure of the GPU is fixed, and it is not possible to make hardware-level changes and optimizations for different neural network structures. In view of their advantages of re-programmability and power efficiency, FPGAs have been considered by researchers for implementing neural network algorithms to accelerate the calculations. In contrast to CPUs or GPUs, using FPGAs can achieve both low power consumption and high performance in one implementation based on the idea of hardware-software co-design. In the study of Zhang et al. [5], they applied optimization methods such as 'loop tiling and transformation' to improve system performance. Finally, they implemented the AlexNet [6] network on an FPGA. This implementation was found to be 17.42 times faster than the CPU implementation, while the power consumption was only 18.61 watt (W).

In recent research on the use of FPGA neural network accelerators, thanks to the High Level Synthesis (HLS) tool designed by Xilinx, the development cycle for designers to use FPGAs for algorithm acceleration is greatly reduced. HLS uses a high-level programming language (C or C++) to generate hardware description language (Verilog or VHDL) code. For the design levels, HLS derives register transfer level (RTL) design from the abstract algorithm level [7]. Our main contribution is a general neural network accelerator architecture that can be freely tailored according to the FPGA hardware resources and its logic elements. The features of the accelerator are determined by the grammatical

characteristics of the C programming. The accelerator computing behaviour is constrained by the advanced extensible interface (AXI) bus in an FPGA development platform. Our other contributions are:

1. Designing and implementing a data acquisition and processing system structure with Xilinx Zynq series chips, where a Convolution Neural Network (CNN) is used for data analysis.
2. Proposing five ways to improve accelerator performance and evaluating the effect of each of these ways.
3. A neural network accelerator structure, which consists of a Process Element (PE) as the smallest computing unit and the AXI bus [8] for the data transmission.

II. SYSTEM STRUCTURE

As shown in Figure 1, the system of data acquisition and processing consists of two parts: a data collector and a neural network accelerator. The accelerator is based on a Xilinx FPGA development board, ZYNQ-7020, in the Xilinx series of ZYNQ-7000. The on-board chip consists of a processing system (PS) and programmable logic (PL). An Ethernet with TCP/IP protocol is applied to transfer data from the collector to the accelerator. The data collector consists of a Raspberry Pi and a camera which is for acquiring image data. The camera could be replaced by other sensors for a different type of data. Since the Raspberry Pi has abundant general purpose input and output (GPIO) resources, the system can also drive actuators according to the output results from the neural network accelerator.

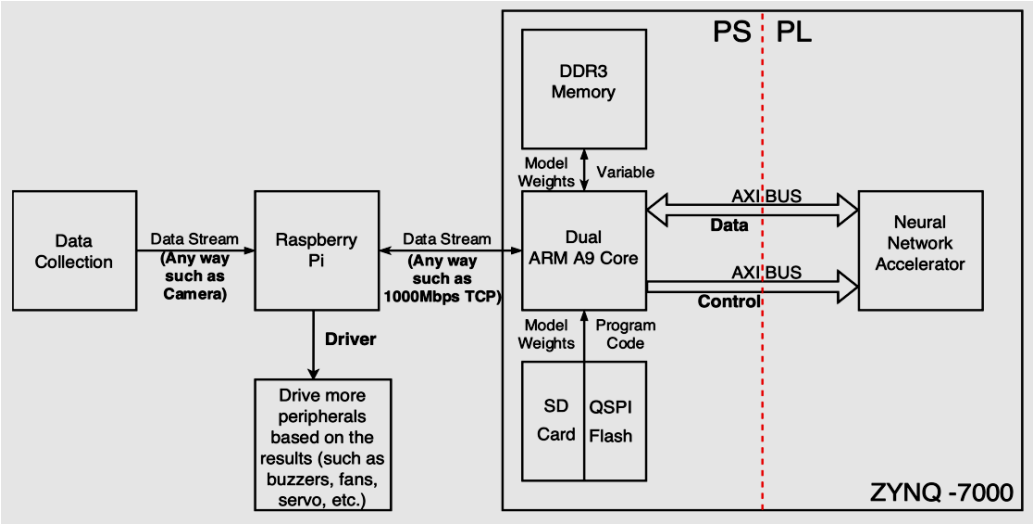

Figure 1. Data Acquisition and Processing System Structure.

The neural network accelerator is deployed in the PL part of the ZYNQ-7020 chip in the form of IP Core, and transmits data through the powerful AXI bus inside the chip and the dual-core ARM A9 processor in the PS part of the ZYNQ-7020 chip. The transmitted data consists of control signals and the data used for processing. The control signals determine the behaviour of the internal PE of the accelerator; for example the size of the feature map in the convolution processing. The internal data is the images and the values of the weights of the layers in the neural network. Output data is the results of the analysis. All the transferred data will be stored in an external DDR3 memory. The data stored in DDR3 will be updated during the data processing. Generally, the size of the ZYNQ's on-chip memory limits the number of neural network weight parameters that can be accommodated. The actual requirement could exceed 0.65MB. In order to accommodate more weight parameters, we put the weights into a SD Card. When the whole system starts to operate, the ARM A9 processor will load the weight parameters stored in the SD Card into the DDR3 memory. The code size of the accelerator design may increase significantly for more complicated applications, so it is stored externally in a QSPI FLASH chip rather than in the ZYNQ on-chip memory.

III. NEURAL NETWORK ACCELERATOR ARCHITECTURE

As shown in Figure 2, the neural network accelerator consists of multiple 'Process Elements' (PE). A PE is the smallest processing unit in the neural network accelerator, and each PE has a particular function. For example, PE-1 in Figure 2 realizes the acceleration of the convolution calculation, and PE-2 realizes the padding acceleration. Each PE has a set of control registers and data registers.

Control registers are used to obtain control signals from the ARM A9 processor on the ZYNQ PS side. Such control signals include the number of convolution kernels, the size of the convolution kernel, the size of the input feature map and other parameters related to the neural network calculations.

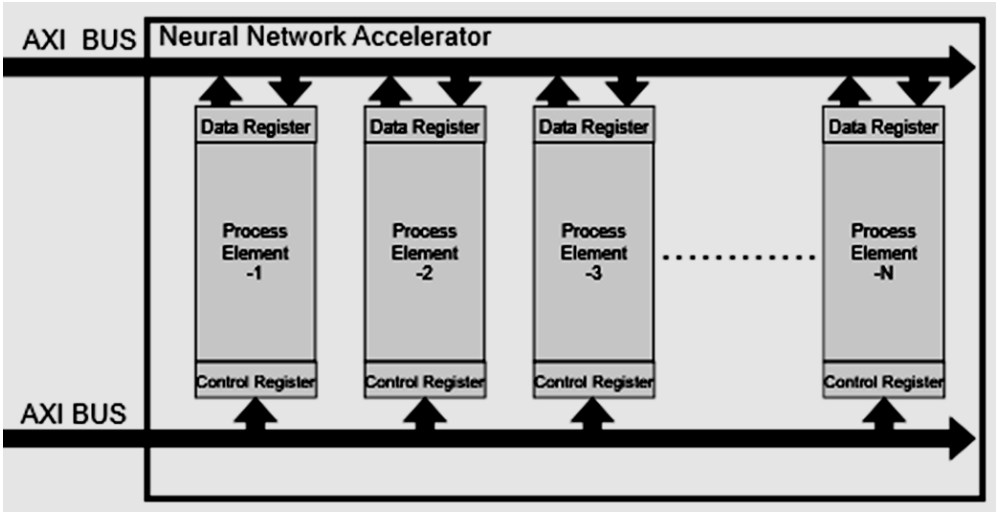

Figure 2. Neural Network Accelerator Architecture.

The data register is used to buffer data to facilitate the pipeline calculation inside the PE. The pipeline is a way of increasing the speed of data processing. The data register design will be explained in detail in section IV.B. Each PE is connected to the PS through the AXI bus to realize data communication. The number of PEs depends on how much of the FPGA resources the designer plans to allocate to the neural network accelerator. For example, one convolutional PE unit in this paper requires 6,254 logic elements each of which is based on a look-up-table (LUT).

To clearly demonstrate our research results, we have applied the PEs to accelerate some common neural networks. Since different neural networks consume different amounts of memory in the PE's internal data registers, we constrained the computation of the neural network so that the neural network can be deployed on the FPGA. The constraints and focus of the acceleration are listed below:

A. Calculation Constraints

- The feature map size is less than or equal to 320 × 320 (*Width* × *Height*), and the number of channels is from 1 to 1024.
- The size of the convolution kernel is 3 × 3 or 1 × 1 (*Width* × *Height*).

B. Operators that have been Accelerated

- Conv2d convolution, Depthwish Conv2d convolution [9].
- Same Padding (Asymmetric Padding is not supported).
- Max Pooling and Average Pooling.
- Fully connected layer calculations.

IV. PROCESS ELEMENT ARCHITECTURE

C. Code Specification

The Xilinx VIVADO HLS converts C/C++ code into Verilog/VHDL code. Different code structures will generate different hardware architectures. For our research, the outer architecture of each PE is the same. For example, every PE has a control register and a data register. Both the control register and the data register are connected to the ARM Cortex-A9 on the ZYNQ PS through the AXI bus. The inner architecture of each PE is different, and it depends on the functions to be implemented by the PE (for example, convolution, pooling, etc.). A PE code specification is given by the pseudo code shown in Fig. 3, which ensures that all PEs have the same outer architecture.

The PE code specification defines that the input of each PE top-level function can only be an address. This brings two benefits:

1. The PE can be addressed quickly in DDR3 memory;
2. The PS can manage the memory of each PE more conveniently.

In the first line of the PE code specification, calculation constraints are established. Lines 2 to 7 of the code give some examples of establishing computational constraints. Lines 8 to 10 establish the parameter type through the variable ‘typedef’. There are two types of parameter. One type is a ‘data parameter’ and the other is a ‘register parameter’.

Algorithm 1 Process Element Code Specification

INPUT: Control register address, (reg_t *)$Register_{addr}$; Weights storage address, (data_t *)$Weights_{addr}$; The address of the input feature map, (data_t *)$Infmap_{addr}$; The address of the output feature map, (data_t *)$Outfmap_{addr}$;

OUTPUT: $NONE$(Directly read and write the input memory address, so the function has no variable output.)

```
1: Set Calculation Constraints;
2: Calculation Constraints Example:
3: {
4: #define MAX_DataReg_Input_fmap_size 3 /*The storage space size of the Input data register*/
5: #define MAX_DataReg_Output_fmap_size 1 /*The storage space size of the Output data register*/
6: etc...
7: }
8: Set Process Element Parameter Type;
9: typedef float data_t; /*Calculation parameter type*/
10: typedef char reg_t; /*Control register parameter type*/
11:
12: function PROCESS ELEMENT(Register_addr, Weights_addr, Infmap_addr, Outfmap_addr)
13:     /*Construct function parameters as AXI bus*/
14:     #pragma HLS INTERFACE m_axi port=Register_addr
15:     #pragma HLS INTERFACE m_axi port=Weights_addr
16:     #pragma HLS INTERFACE m_axi port=Infmap_addr
17:     #pragma HLS INTERFACE m_axi port=Outfmap_addr
18:
19:     PE_Function_Block{
20:     Access the address input by the function, and read and write the data in the address according to the PE function to be realized;
21:     }
22:
23:     return NONE
24: end function
```

Figure 3. Process Element Code Specification

Data parameters store the input, intermediate values and the resulting values during the PE calculation. Register parameters are responsible for controlling the process during the PE calculation. Such design gives the capability to quickly adjust the resource consumption of PE calculations. Lines 14 to 17 of the code feed the input variables of the PE function into the AXI bus, and transmit the data through the AXI bus. Lines 19 to 21 define the PE function block, which determines the function of the PE.

D. Performance Improvement Methods

The above discussion proposed a flexible neural network accelerator architecture. However, for practical applications, the performance still needs to be further tuned. In this section, five methods are proposed to improve the performance of each PE, which eventually improves the overall performance of the accelerator. The five methods are as follows:

a) Increasing the input clock frequency of the PE.

When the clock frequency was 100 MHz, its inference speed was 113 Frames Per Second (FPS). Increasing the clock frequency to 250 MHz increased the PE speed to 282 FPS. The computational speed of the PE was found to be almost linearly dependent on the clock frequency as shown in Fig. 4.

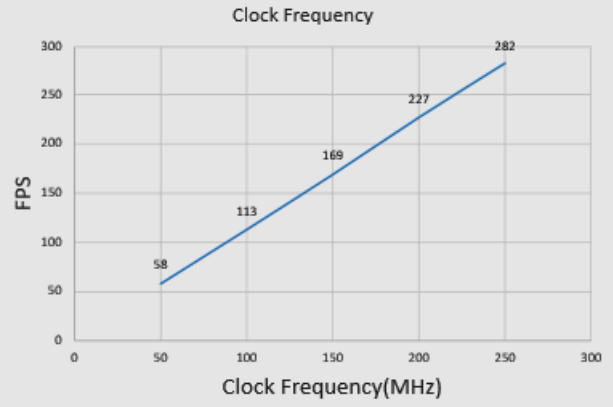

Figure 4. PE Frames per Second vs Clock Frequency.

b) Enabling the Data Cache of the ARM A9 processor.

The data cache will speed up the data transfer rate between the ARM Cortex-A9 processor and the PEs. The process is actually mapping access to DDR3 memory addresses. Fig. 5 shows that an improvement of 18.5% in PE performance was achieved, in terms of FPS, by enabling the Data Cache.

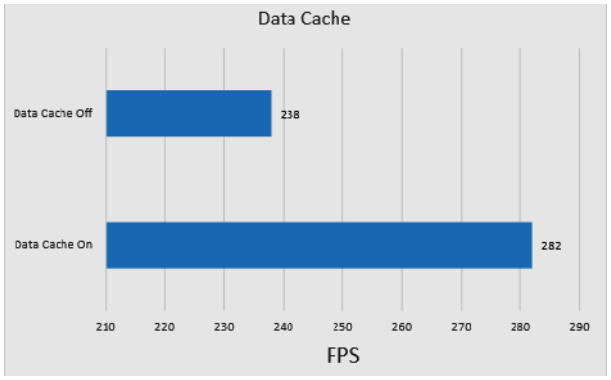

Figure 5. Impact of Data Cache on PE Performance.

c) Quantising parameters and performing parameter fusion.

Embedded systems and FPGAs commonly have low computational performance and limited RAM. If the data type of the neural network weights is floating point, for example *float32*, this may take up too much computing resources and reduce the inference speed. To increase the inference speed, quantised fixed point weights may be used, though this will be at the expense of some reduction in the accuracy of the neural network. The performace of different fixed point quantisation schemes is demonstrated in Fig. 6 and Fig. 7.

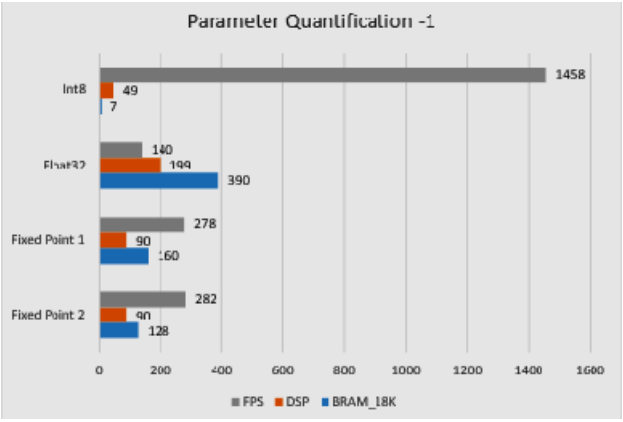

Figure 6. Quantisation vs Resource Consumption of DSP and BRAM_18K (Fixed Point 1: 16 bit Q12 format; Fixed Point 2: 16 bit Q8 format.)

As shown in Fig. 6, a processing speed of 1458 FPS was achieved using 'int8' quantisation (signed 8 bits per value). This compares with 140 FPS for float32, 278 FPS for int16 with Q12 format and 282 FPS for int16 with Q8 format. Further, the use of 'int8' reduced the consumption of computational resources of 49 DSP units and 7 BRAM_18K. to 40 FPS (for DSP) and 7 FPS (for BRAM_18K). This compares with 199 and 190 FPS (DSP and BRAM_18K respectively) for float32, 90 and 150 FPS (DSP and BRAM_18K) for int16 with Q12 format and 90 and 128 FPS for int16 with Q8 format. Therefore there is a very significant speed improvement together with a very significant reduction computational resources to be gained by using 'int8' quantisation.

Fig.7 presents FPS performance improvement under the FPGA resource of LUT and flip-flop register (FF). The idea of quantisation can also be implemented using Convolution layer and Batch Normalization [10] where parameter fusion can be applied [11].

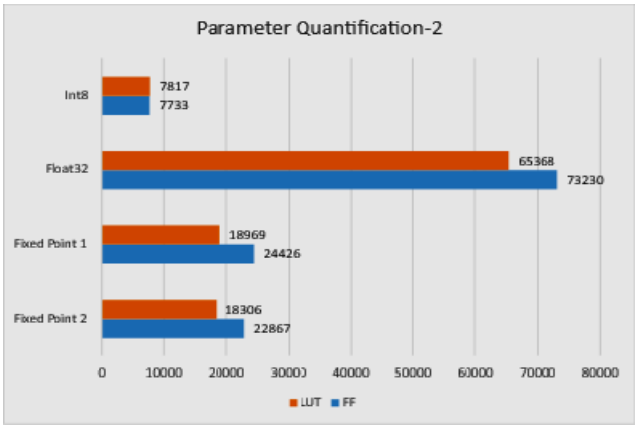

Figure 7. Quantisation vs Resource Consumption of LUT and Flip-Flop Register (Fixed Point 1: 16bit Q12 format; Fixed Point 2: 16bit Q8 format.)

d) Performing the Entire Calculation within a Single PE.

To further improve computational speed, the calculation of the entire neural network can be performed within a single PE, and the neural network weights can be stored in the Block RAM (BRAM) of the FPGA. The speed of accessing the BRAM will be much faster than accessing DDR3 memory through the AXI bus. Since the BRAM size varies greatly among different types of FPGA, the designer needs to ensure that the weights of the entire neural network occupy a memory size less than or equal to the BRAM size. The internal BRAM size of the ZYNQ-7020 FPGA used in our research is 0.625 MBytes. We used the proposed method to accelerate a small 'character recognition' network, illustrated in Fig. 8, which contains 79,242 parameters. An open source of 'Chars 74K' dataset was used for the PE performance evaluation. Characters with three computer typing fonts, normal, italic and bold, were extracted from the dataset. These characters in the dataset were divided into 62 classes (0-9, A-Z, a-z). Each class could be normal, italic or bold font. In our experiment, ten classes of characters 0-9 were used. The final extracted dataset used for testing the accelerator consisted of 10,160 single-channel images of 128 by 128 pixels, where there were 1016 images for each class. Its resource consumption and performance are shown in Fig. 9. A sample of the handwritten numbers that were recognised successfully in real time is shown in Fig. 10.

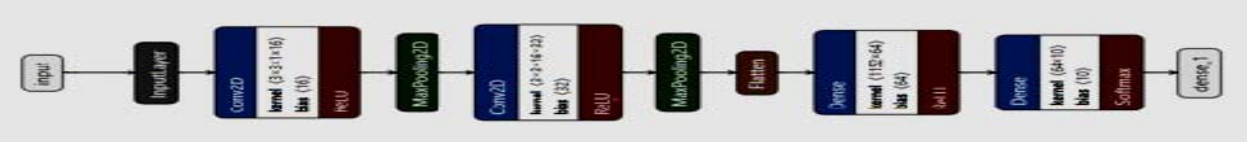

Figure 8. Neural Network Structure

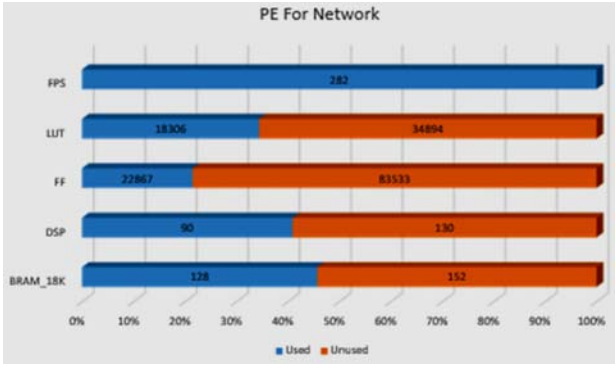

Figure 9. PE For Network Resource Consumption of FPGA Resources.

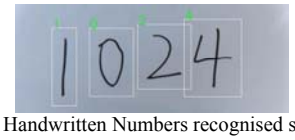

Figure 10. Handwritten Numbers recognised successfully

e) Implementing data register and pipelined operation within a single PE [12].

Although it is possible to achieve PE-level pipeline acceleration by using multiple PEs for multiple calculation functions, the operations involved in each function are still taken sequentially within each PE. Therefore, to further improve the performance of each PE, pipeline calculation may be implemented inside the PEs. The PE operations can be summarized as three steps:

1. Read the address of the control register, weights and input data from DDR3 memory.
2. Run the calculation and obtain the result.
3. Write the result to DDR3 memory.

Steps 1 and 3 are to input data from and output data to DDR3 memory, which cause the PE to spend a lot of time on data transferring. If the time of data transferring is shortened, the performance of the PE can be improved. After implementing the pipeline in a PE, operations of input/output data and algorithm calculation can be parallelized within a single clock cycle. Pseudo code for the implementation is shown in Fig. 11.

Algorithm 2 Data register and pipeline of PE

INPUT: Pipeline start flag, $Pipeline_s$; Pipeline control signal, $Pipeline_c$; Input data to the data register, $Input_d$; Index position row, i; Index position col, j;
OUTPUT: Output data to the data register, $Output_d$;

```
Pipeline_s ← 1
Pipeline_c ← 1
i ← 0
j ← 0
data1 ← 0 /*Data register 1*/
data2 ← 0 /*Data register 2*/
if Pipeline_s == 1 then
    Read the data1 at index position [i,j] of Input_d.
    Pipeline_s ← 0
end if
for i = 0, j = 0 → all_data_index_length do
    if Pipeline_s == 1 then
        Read the data2 at index position [i, j+1] of Input_d.
    else
        Read the data1 at index position [i,j+1] of Input_d.
    end if
    if Pipeline_s == 1 then
        Calculate the data1 at index position [i, j].
        Save the data1 result to index position [i, j] of Output_d.
        Pipeline_s ← 0
    else
        Calculate the data2 at index position [i, j].
        Save the data2 result to index position [i, j] of Output_d.
        Pipeline_s ← 1
    end if
end for
return Output_d;
```

Figure 11. Pseudo Code of Data Register and Pipeline.

To illustrate the performance improvement of the implementation, the calculation of a convolution PE may be visualized. Figure 12 shows the typical behaviour of a convolution PE without parallelization (kernel size = 3 × 3, stride = 1). Figure 13 demonstrates the typical behaviour of a parallelised PE convolution, where the kernel size is 3 × 3 and the stride equals 1. It can be observed that the convolution PE can load, calculate and store two different sets of data within one operation cycle.

This illustrates how the parallelization improves the efficiency of the data processing. The parallelisation is illustrated at register level as shown in Figure 14, where the *Data1* and *Data2* registers alternately perform calculations for the entire Feature Map. Table 1 provides the states of the parameters for the first 5 operation cycles.

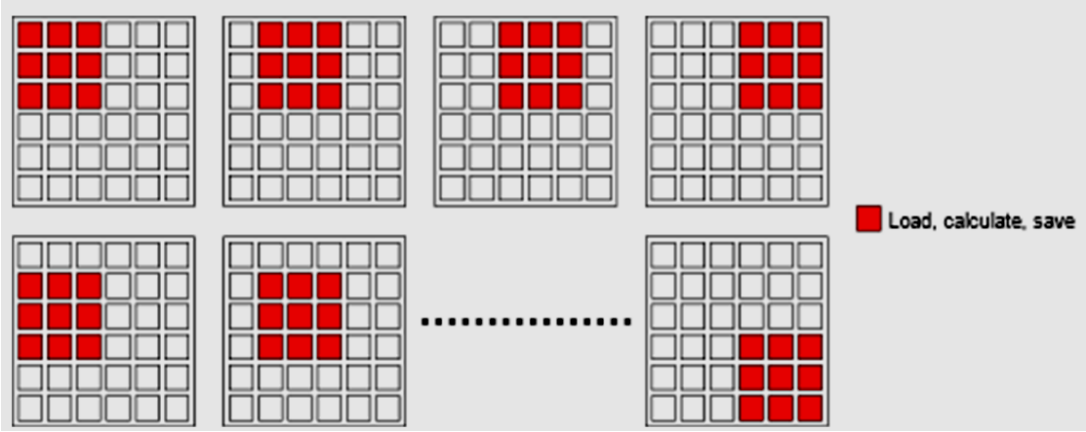

Figure 12. The Process of Convolution PE without Parallelisation.

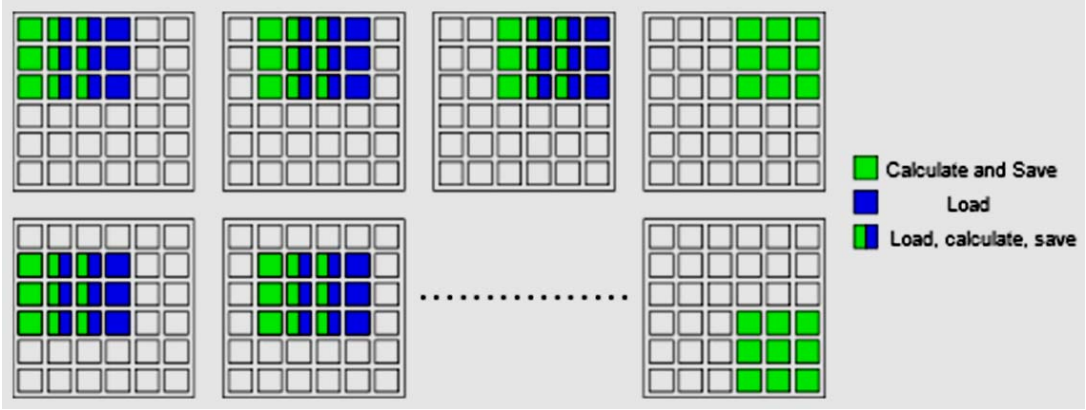

Figure 13. The Process of Convolution PE with Parallelisation.

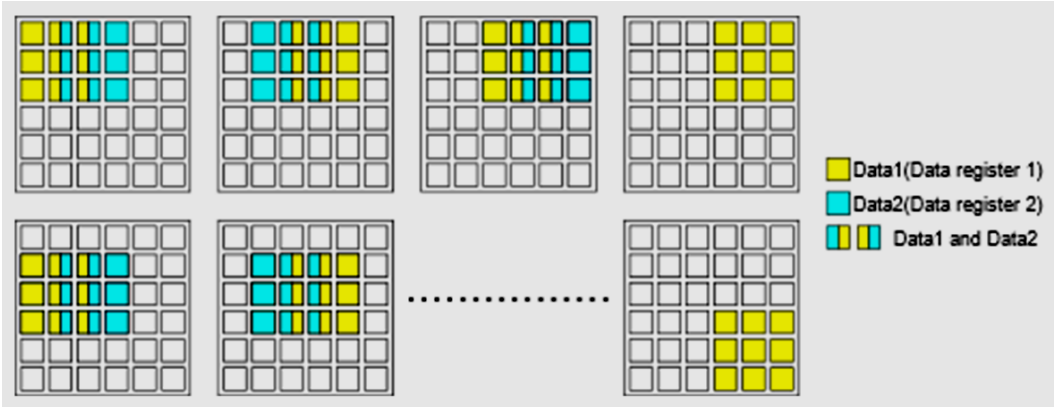

Figure 14. Parallelization of Convolution PE at Register Level.

TABLE. 1. STATES OF THE PARAMETERS FOR THE FIRST FIVE OPERATION CYCLES.

Initialization	PE calculation process parameter changes					
Pipeline_s	1	0	1	0	1	…
Operate: D1_L	D2_L, D1_C	D1_L, D2_C	D2_L, D1_C	D1_L, D2_C	D2_L, D2_C	…
Output Data Index: [0, 0]	[0, 1], [0, 0]	[0, 2], [0, 1]	[0, 3], [0, 2]	[0, 4], [0, 3]	[0, 5], [0, 4]	…
NOTE: D1_L=Data1 Load; D1_C=Data 1 Calculate; D2_L=Data2 Load; D2_C=Data2 Calculate.						

It is not advisable to parallelise all PEs even though this could potentially improve the processing speed. Parallelising all PEs may lead to more FPGA logic elements (IE) being required than are available. Since FPGA logic element resources are usually limited, parallelisation should be prioritised for PE convolution calculations that consume a lot of FPGA resources [13]. Figure 15 below shows the impact of parallelisation on the calculation speed and resource consumption of a convolution PE.

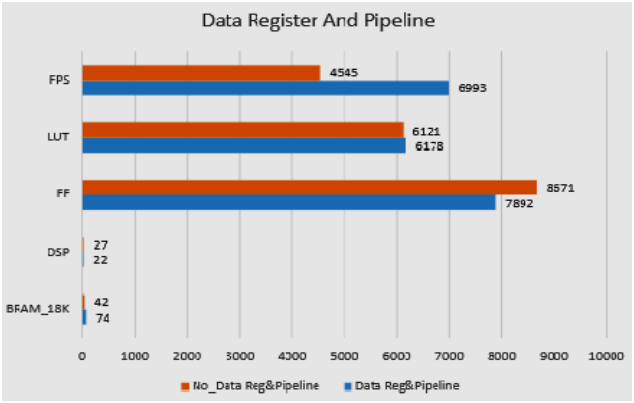

Figure 15. Performance of Data Register Pipelining vs Resource Consumption.

V. EVALUATION

A. Resource Consumption

The PE consumption of BRAM_18K and DSP resources is illustrated in Figure 16.

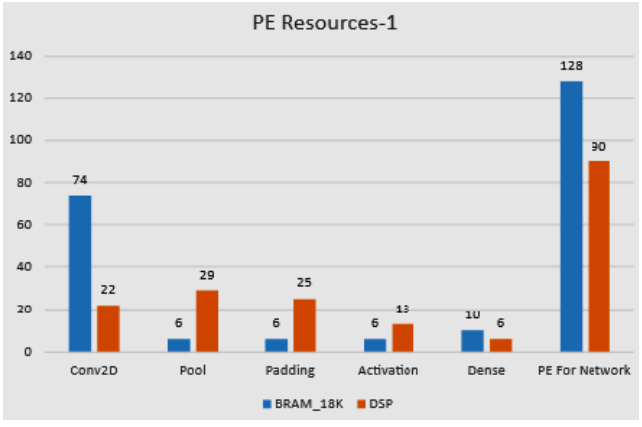

Figure 16. PE Resource Consumption

Figure 17 illustrates the consumption of flip-flop (FF) and look-up table (LUT) resources. It is observed that the overall resource consumption in parameterising PE calculation functions is much higher than is consumed for implementing the entire neural network in a PE for Network. For example, 29,144 LUTs are consumed in a parameterised PE, while 18,306 LUTs are used in a PE for Network.

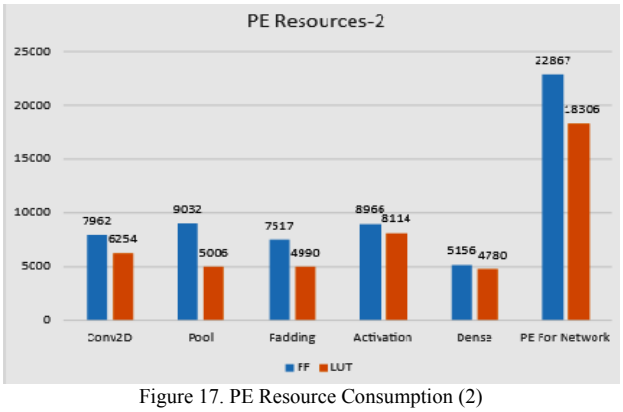

Figure 17. PE Resource Consumption (2)

B. Computing Performance

Figure 18 compares the performances of different inference implementations of the same neural network whose structure is shown in Fig. 8. The GPU inference with *TensorFlow 2* appears to have the lowest computing performance with a processing speed of 18 FPS. The *Keras* package was used to test the inference speed of *Tensorflow 2*.

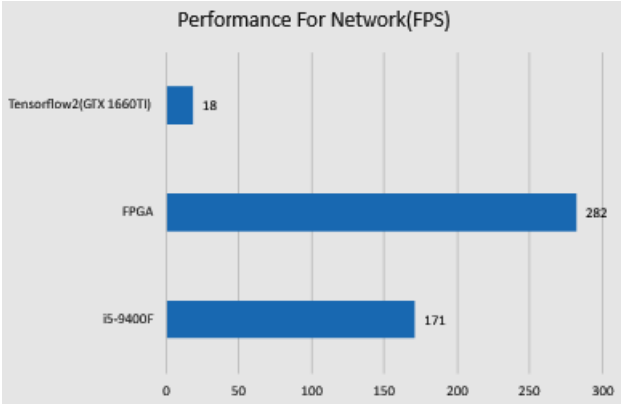

Figure 18. Inference Performances on Three Platforms

It proved to be very inefficient without any code optimization. A CPU running C-code achieved a processing speed of 171 FPS which is much better than was achieved with the GPU implementation. The computing performance based on PE for Network and implemented with a FPGA was found to be 15.6 times faster than the GPU implementation, and 1.6 times faster than the CPU implementation.

Figure 19 lists the computing performance of different functional PEs. By comparing the performance of PEs between CPU and FPGA, we can see that the performance of FPGA and CPU is very similar for convolution and padding calculation.

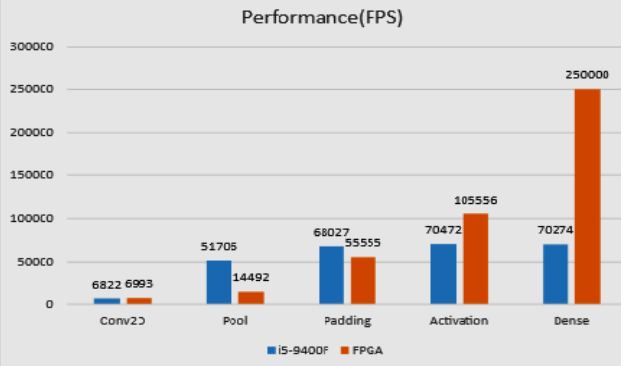

Figure 19. Performance of Different Functional Pes.

For Pooling, the algorithm we used requires a lot of loops and traversal calculations, whose operation is very memory bandwidth intensive. Since we did not apply the parallelization in section IV.B to Pooling PE, the performance is much lower compared to that of CPU. For Activation and Dense calculation, the performance of FPGA is higher than that of CPU.

C. Energy Consumption Comparison

A comparison of the power efficiencies of the different implementations is given in Figure 20 in terms of FPS per watt.

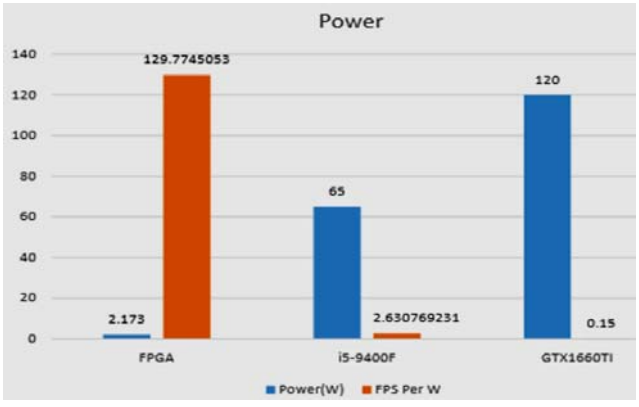

Figure 20. FPS Per Watt

The value of the FPGA power consumption was obtained from the Xilinx Vivado real-time power data output. The CPU and GPU power consumptions were defined by their Thermal Design Power (TDP) data [14], which was provided by the manufacturers.

VI. CONCLUSION

The FPGA Neural Network Accelerator proposed in this paper has been shown to offer greater computational performance compared to consumer-grade CPU and GPU designs and had much lower power consumption and higher efficiency. The proposed design can be widely used in the field of embedded edge computing for flexible and extensible neural network computation. Future work can be focused on more neural network types.

In many neural network accelerators, the architecture is fixed for a specific application. When the neural network application is changed, the number of convolutional window slides, hence the size of the convolutional kernel and other computational parameters inside the accelerator, need to be redesigned to have the effect of acceleration, which means a different architecture of an accelerator needs to be designed. This research has implemented a general architecture for the accelerator, where the computational parameters are configured as registers which are connected to an ARM Cortex A9 processor via the AXI bus. When the neural network structure is changed, there is no need to modify the accelerator architecture. The user can simply reset the registers via the ARM Cortex A9 processor to adapt to new neural network.

In addition, the general accelerator architecture implemented by FPGA has very low power consumption. For a specific neural network application, an I5-8400 CPU implementation consumed a power of 65 watts, while the corresponding FPGA implementation consumed only 2.173 watts. The design of the accelerator and the implementation of the FPGA is applicable for 'Internet of Things' (IoT) edge computing systems which may rely on batteries as power supplies.

ACKNOWLEDGMENT

Further research will be carried out and evaluated on a more powerful FPGA kit, which is funded by 'Research England'. We acknowledge the support provided.

REFERENCES

[1] Atzori, Luigi, Antonio Iera, and Giacomo Morabito. "The internet of things: A survey." Computer networks 54.15 (2010): 2787-2805.

[2] Shi, Weisong, et al. "Edge computing: Vision and challenges." IEEE internet of things journal 3.5 (2016): 637-646.

[3] Bogoslovskii, Ivan A., et al. "Implementation of an Approximator Based on a Multilayer Perceptron and Wavelet-Neural Network on the STM32 Microcontroller." 2020 IEEE Conference of Russian

Young Researchers in Electrical and Electronic Engineering (EIConRus). IEEE, 2020.

[4] Jose, Edwin, et al. "Face recognition based surveillance system using facenet and mtcnn on jetson tx2." 2019 5th International Conference on Advanced Computing & Communication Systems (ICACCS). IEEE, 2019.

[5] Zhang, Chen, et al. "Optimizing fpga-based accelerator design for deep convolutional neural networks." Proceedings of the 2015 ACM/SIGDA international symposium on field-programmable gate arrays. 2015.

[6] Krizhevsky, Alex, Ilya Sutskever, and Geoffrey E. Hinton. "Imagenet classification with deep convolutional neural networks." Advances in neural information processing systems 25 (2012): 1097-1105.

[7] Meeus, Wim, et al. "An overview of today's high-level synthesis tools." Design Automation for Embedded Systems 16.3 (2012): 31-51.

[8] AMBA AXI Protocol Specification, Sunnyvale, CA, USA:Axis, 2003

[9] Howard, Andrew G., et al. "Mobilenets: Efficient convolutional neural networks for mobile vision applications." arXiv preprint arXiv:1704.04861 (2017).

[10] Ioffe, Sergey, and Christian Szegedy. "Batch normalization: Accelerating deep network training by reducing internal covariate shift." International conference on machine learning. PMLR, 2015.

[11] Jung, Wonkyung, et al. "Restructuring batch normalization to accelerate CNN training." arXiv preprint arXiv:1807.01702 (2018).

[12] Ramamoorthy, Chittoor V., and Hon Fung Li. "Pipeline architecture." ACM Computing Surveys (CSUR) 9.1 (1977): 61-102.

[13] Burrus, C. Sidney, and T. W. Parks. "Convolution Algorithms." Citeseer: New York, NY, USA (1985).

[14] John L. Hennessy; David A. Patterson. "Computer Architecture: A Quantitative Approach (5th ed.), " Elsevier. p. 22. ISBN 978-0-12-383872-8 (2012).