International Journal of Advanced Technology and Engineering Exploration, Vol 11(119)
ISSN (Print): 2394-5443 ISSN (Online): 2394-7454
http://dx.doi.org/10.19101/IJATEE.2023.10102538

Research Article

Efficient implementation of artificial neural networks on FPGAs using high-level synthesis and parallelism

Mini K. Namboothiripad[1*] and Gayathri Vadhyan[2]
Department of Electrical Engineering, Agnel Charities Fr. C. Rodrigues Institute of Technology, Vashi Navi-Mumbai, Maharashtra, India[1]
School of Electrical Engineering, Vellore Institute of Technology, Vellore, Tamilnadu, India[2]

Received: 01-December-2023; Revised: 12-October-2024; Accepted: 16-October-2024

Abstract
Artificial neural networks (ANNs) have gained significant attention for their ability to solve complex problems in various domains. However, the efficient implementation of ANN models on hardware remains challenging, particularly for systems requiring low power and high performance. Field programmable gate arrays (FPGAs) offer a promising solution due to their reconfigurability and parallel processing capabilities. This study explores the implementation of ANN on an FPGA using high level synthesis (HLS), focusing on optimizing performance by leveraging weight-level and node-level parallelism. Two methodologies were proposed for efficiently implementing ANN computations on an FPGA. The focus was on partitioning the computations of the ANN's first layer to the programmable logic (PL) of a system-on-chip (SoC) FPGA, while offloading the processing of subsequent layers to a 666 MHz, advanced reduced instruction set computer machine (ARM) processor. Six designs with varying levels of weight-level and node-level parallelism were implemented on a python based FPGA (PYNQ) board. Multiple processing elements (PEs) and sub-PEs were instantiated in the PL to extract parallelism from the ANN computations. Single-precision floating-point accuracy was used throughout the implementations. The custom digital design, operated at 150 MHz, achieved a significant speedup, demonstrating 2.5 times faster computation than the 666 MHz ARM processor for the entire ANN computation even with the limited resources available on the PYNQ board. Scaling up with multiple FPGAs could result in performance levels comparable to generic processors. The integration of HLS and the control block redesign capabilities of the ARM processor made the system adaptable to various applications without requiring extensive knowledge of hardware descriptive languages (HDL). This research shows that FPGA-based implementations of ANN, especially using HLS, offer a viable and efficient alternative to graphical processing unit (GPU) or processor-based designs for ANN applications. The demonstrated speedup achieved through parallelism and the use of PL indicates the potential of FPGAs in creating dedicated application-specific integrated circuits (ASICs), for ANN applications, offering a competitive option compared to traditional GPU or processor-based solutions.

Keywords
Artificial neural network, Field programmable gate arrays, High level synthesis, Node and weight level parallelism, Programmable logic.

1.Introduction
Machine learning algorithms form a major part of artificial intelligence and are widely utilized everywhere to perform various tasks that have become a part of our daily routine [1]. They are used in various domains such as business intelligence, medicine research, image processing, and textual analysis [2]. The major utilization of these machine learning algorithms is in the processes of prediction and classification.

Artificial neural network (ANN) is a subfield of machine learning and has the potential to learn and make important decisions on its own [2].

Software implementation of such algorithms has become simpler due to the introduction of libraries such as Scikit Learn, TensorFlow, etc [1]. As machine learning algorithms become increasingly complex to achieve higher accuracies, their computational demands escalate. This often necessitates a separate graphical processing unit (GPU) to achieve the required processing speeds,

*Author for correspondence

introducing numerous challenges for real-time implementation [3].

In recent years, field programmable gate arrays (FPGAs) have emerged as a successful accelerator for machine learning algorithms [4]. Their versatility for prototyping diverse algorithms and ability to parallelize computations effectively, make them a preferred choice for accelerating such processes. FPGAs are semiconductor devices that consist of input/output (I/O) blocks or pads, configurable logic blocks, and programmable routing or interconnects [5]. Configurable logic blocks implement basic combinational and sequential logic, provide storage, and are connected to the external components using I/O blocks. The connection between logic blocks and I/O blocks is established using programmable routing. They have a huge set of embedded components like digital signal processor (DSP) blocks (used to multiply and accumulate), flip-flop (FF), look up table (LUT), etc.

FPGAs are reprogrammable and can be used for desired application or functionality requirements. Since FPGAs are reprogrammable, highly cost-effective in the long run, and can be used for prototyping purposes [5]. Another important advantage of FPGA is that it can be designed to process data parallel by instantiating multiple blocks. FPGAs can exploit the available parallelism in ANN. Also, it can offer semi-custom systems for neural networks which can be easily adaptable with the new technological developments [4]. It is observed that with proper optimization, the performance of FPGA can be made superior to that of embedded GPU [6–8]. However, the development process for the FPGA is more tedious and time-consuming [9].

Neural networks offer multiple avenues for parallelism, encompassing training, layer, node, and weight levels [3]. In layer parallelism, FPGA implementation can leverage pipelining to process layers concurrently [10, 11]. Available node-level parallelism, due to the large number of neurons or nodes, can be exploited by first multiplying the inputs with different columns of weights in parallel and then using tree adders to accumulate the products [12]. Likewise, weight-level parallelism can be harnessed by concurrently processing various input characteristics with neuron weights. The computations can be done in parallel by partitioning input and weight matrices into rows, with the final output synthesized by concatenating outputs from each parallel block [13].

Challenges for the implementation of ANN in FPGA include the requirement of a large number of computational resources, external memory bandwidth, and a significant amount of storage [14]. For a large neural network, if the resource sharing is not optimized, then the entire implementation may not fit on the FPGA due to limited resources. If the storage is not sufficient, then the weights must be stored in an external memory and then during computation, it is to be transferred to the FPGA [15].

Leveraging available parallelism and optimizing operation sequencing in neural network implementations on FPGAs are heavily influenced by data transfer efficiency and resource utilization [14, 15]. By efficiently reusing resources for computation and storing the partial results in internal memory, the challenge of resource utilization can be substantially mitigated [16]. Techniques such as layer multiplexing, where only the largest layer is instantiated and utilized for implementing other layers with a control block, aim to optimize resource utilization. However, there is often a trade-off between reduced resource requirements and moderate speed overhead. Moreover, training ANNs for specific tasks, presents additional challenges, with offline training being explored to enhance scalability [17].

The integration of hardware and software components through co-design approaches have shown promise in mitigating FPGA resource limitations. Combining hardware and software solutions in design offers significant speed enhancements compared to dedicated hardware or software implementations [18].

Recognition of handwritten content across various languages using ANNs stands out as a prominent field of research [19]. However, the choice of arithmetic representation, whether fixed or floating point, impacts both accuracy and resource utilization, with floating point representations typically offering higher accuracy at the cost of increased resource usage and computational time [16]. Despite these challenges, optimizing FPGA implementations can yield significant speed improvements compared to traditional processing units like advanced reduced instruction set computer machine (ARM) processors, even when maintaining floating-point accuracy [20].

This paper presented a hardware and software co-design approach using a python based system on chip (SoC) A Python based FPGA (PYNQ) for

implementing an ANN, tasked with classifying handwritten digits. The ANN is initially trained offline, and subsequently, the parameters obtained are utilized in the hardware implementation to finalize the classification. To optimize resource utilization, the largest layer of the ANN is implemented by instantiating multiple processing elements (PE) within the programmable logic (PL) of the SoC-FPGA. Within each PE, multiple sub-PEs are instantiated to extract available node level and weight level parallelism. The remaining layers, along with control logic that manages input generation and communication with the PEs, are handled by the ARM processor in the SoC.

Custom-designed hardware intellectual property (IP) cores, featuring varying degrees of node and weight-level parallelism, are created and evaluated for performance. Despite the PYNQ board's limited resources, the design achieved up to a 2.5x speedup compared to running all computations on the ARM processor, while maintaining floating-point precision. This approach is also scalable, requiring only a redesign of the ARM processor's control blocks to adapt it to different applications, making it more flexible than other solutions in the literature.

The novelty of the paper is the systematic approach adopted for the design and implementation of the ANN acceleration algorithm on FPGA, focusing on tools, methodologies, and architectures for hardware and software co-design. Also, instead of using hardware descriptive languages (HDL), various hardware designs were developed in C++ using high level synthesis (HLS) which converts C/C++ code into register transfer level (RTL) code. This approach allows the designs presented in the paper to be easily replicated and adapted for various applications without requiring extensive expertise in HDL or hardware development.

This paper is structured as follows: section 2 is the literature review, and section 3 explains the methodologies explored for the ANN implementation on FPGA using HLS. Section 4 contains results which is followed by discussions and conclusions.

2.Literature review

Machine learning algorithms, a core component of artificial intelligence, are widely used for tasks like prediction and classification across domains such as business, medicine, and image processing [1, 2]. As these algorithms grow more complex, their computational needs increase, often requiring GPUs, which can pose challenges for real-time implementation [3]. FPGAs have emerged as effective accelerators for machine learning due to their re-programmability, ability to parallelize computations, and cost-efficiency for prototyping. Their architecture allows them to exploit parallelism in neural networks, making them adaptable to evolving technologies and ideal for high-performance applications [4, 5].

The performance of FPGA and embedded GPU are compared in papers [6–8] and observed that with proper optimization, the performance of FPGA can be made superior to that of embedded GPU [7]. However, the development process for the FPGA is more tedious and time consuming [6–9].

Various materials are available in the literature for the effective implementation in FPGAs to exploit different stages of parallelism in an ANN, which includes training parallelism, layer parallelism, node parallelism, and weight parallelism [10–14]. Operational challenges for the implementation of ANN in FPGA, mainly include resource constraint and external memory bandwidth [15, 16].

Different optimized hardware implementation strategies have been explored in the literature for ANN implementations on an FPGA for various applications [17–20]. The layer multiplexing method, which exploits the sequential processing of the layers in a neural network, is presented in [21]. In this approach, only the single neuron layer is physically implemented in parallel and reused multiple times to simulate multi-layer architectures. The appropriate activation function, input, and weight functions of each computing layer are ensured by the control block. However, while multilayer networks have been implemented for different network configurations it was observed that the resources required are reduced considerably for any given application with the cost of moderate speed overhead.

Several optimization strategies have been explored in the literature to reduce resource requirements without compromising much on accuracy and computational speed. Hardware implementation of an ANN model using FPGA was developed in [22] using various activation functions. A hardware-optimized activation function has been proposed in [14] to minimize resource usage while improving computational accuracy on FPGA, with no impact on training accuracy. Another approach reduces I/O requirements by reusing input data [16]. A hardware

architecture aimed at increasing computational efficiency per DSP and adaptable to various FPGA sizes is discussed in [23]. Additionally, the design of posit multipliers is more efficient, as compared to IEEE-754 multipliers in terms of area and speed, as proposed in [24, 25]. Also, the use of stochastic computing to reduce the area and power utilization with an increase in speed and accuracy is explored in [26].

Literature shows that the limited resource problem in FPGA can be reduced using optimization of hardware and software co-design [27], besides multiplexing. In [28], direct torque control method implementation for the induction motor on FPGA is implemented with hardware and software co-design, which integrates both hardware and software parts on the same chip. The results clearly showed that this approach outperforms any method based on either hardware or software alone. Apart from their application with ANN, literature extensively explores diverse methodologies and optimization techniques employed in the development of FPGA-based convolutional neural network accelerators [29, 30].

Handwritten recognition in various languages using ANN is a notable area for research [31–37]. Acceleration for the recognition of handwritten digits using FPGA was also proposed using Vivado HLS [31] but with a fixed-point representation for the implementation. The impact of arithmetic representation, fixed to floating point representation, on the FPGA, is analyzed in [38-40], which clearly shows more accuracy with floating point representation but with the burden on resources and computational time.

Numerous studies have demonstrated that with proper optimization, FPGAs can outperform embedded GPUs in terms of performance, although FPGA development typically requires more time. The literature also highlights strategies for efficient FPGA implementation of ANNs, utilizing parallelism, multiplexing, custom multipliers, and hardware and software co-design to address challenges like resource limitations and memory bandwidth, with careful consideration of how arithmetic representation affects both accuracy and resource consumption. In this approach, by leveraging FPGA with floating-point precision, a speedup of 2.5x could be achieved for ANN implementation on FPGA compared to executing the computation solely on the ARM processor. Additionally, the proposed solution is scalable, requiring only a redesign of the ARM processor's control blocks to adapt to different applications, offering more flexibility than many existing methods.

3.Methods

Various methodologies are presented to implement an ANN model on FPGA to classify handwritten digits, which were first trained in Python. With the help of the parameters obtained after training the ANN, different hardware designs are developed and analyzed. These designs are created with the help of C++ code using Vivado HLS tool, which is then converted to form IP. The generated IPs are then integrated with other IPs available in Xilinx's IP integrator, Vivado HLx, to create the required system.

The proposed designs with 150 mega hertz (MHz) clock frequency on the PYNQ FPGA board are evaluated by analyzing their FPGA resource utilization and the overall computation latency. Additionally, the latency encompassing all computations on the 666MHz processor is computed, allowing for a comprehensive assessment of each design's acceleration. This acceleration is derived by comparing the latency ratio of each design to the baseline latency.

3.1The ANN model

The model is first trained in Python using Tensorflow and Keras, after which the parameters obtained are used to create hardware, for the prediction of handwritten digits on a PYNQ board using Vivado HLS. The Modified National Institute of Standards and Technology database (MNIST) dataset was used [41, 42] to train and test the model in Python. It consists of 70000 greyscale images of handwritten digits of size (28,28), out of which 60000 is for training and 10000 for testing. The model has three hidden layers, the first layer has 128 neurons, and the second and third hidden layers have 24 neurons. The input matrix is unrolled into a vector of size 784. The output layer has ten neurons, ranging from values 0-9. The hyperparameters for training the neural network are the number of epochs which is 50, the batch size, which is 64, the learning rate, which is 0.01, and the optimization algorithm used is Adam. The activation function for the hidden layers is a rectified linear unit (ReLU), while a sigmoid function is for the output layer.

The model is a fully connected neural network with weights and biases. A dropout layer and a batch normalization layer are also present after every

hidden layer. The usage of a dropout layer while training helps in reducing the possibility of the model overfitting. The batch normalization, after the activation of every hidden layer, helps to normalize the values before they enter the next layer.

3.2Different methodologies used for hardware designs

Two different methodologies are explored in this paper to analyze the computational performances of the hardware implementation of the ANN with respect to resource utilization and latency. In both cases, a PE is constructed for the first layer, which has *n* sub-PEs (SPE1-SPEn). In methodology-1, multiple PEs in the PL exploit the node-level parallelism, while the sub-PEs exploit the available weight-level parallelism in layer-1 for the ANN. On the other hand, in methodology 2 multiple PEs are instantiated to extract the weight-level parallelism and the sub-PEs for the available node-level parallelism.

3.2.1Designs implemented using methodology-1:

Four different designs are implemented using this methodology by varying the number ('n') of sub-PEs within the PE, thus obtaining various degrees of weight level parallelism. The number of parallel PEs in the PL also varied between the different designs, depending upon the available resources in the PL. However, the total computation is distributed amongst eight PEs for all the designs.

The complete layer-1 weight matrix of dimension 784×128 is column partitioned into eight equal parts of dimensions 784×16 (represented as W_1-W_8) to make it individually available for each of the eight PEs, to exploit the node-level parallelism. Each PE takes the corresponding column partitioned weight matrix (W_j) and the input layer matrix (represented as I_p) as the input.

Within the PE, the input layer matrix (I_p) of size 784 is partitioned into 'n' equal parts (represented as I_{p1}-I_{pn}), where the value of 'n' is the same as the number of sub-PEs instantiated within the PE. Also, the column partitioned weight matrix W_j available in the PE is further partitioned row-wise into W_{1j}-W_{nj}, which is then fed into the sub-PEs (SPE1-SPEn) along with the corresponding part of the input matrix (I_{p1}-I_{pn}). Here, the number 'n' also represents the weight level parallelism exploited. The division of the complete layer-1 weight matrix is shown in *Figure 1*.

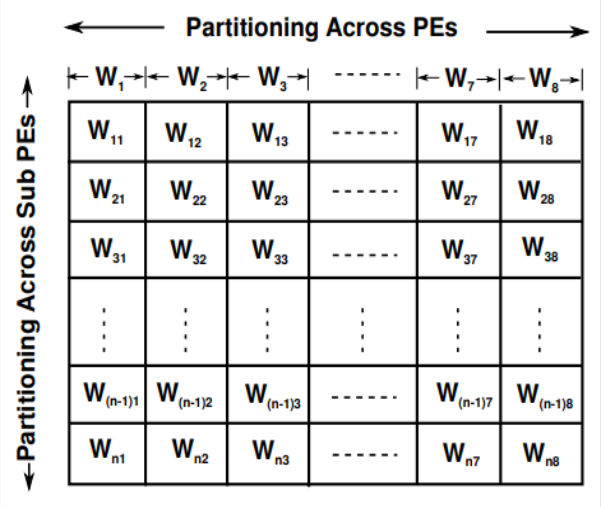

Figure 1 Division of weights across eight PEs and n sub-Pes

The division of the column partitioned weight matrix W_j and input matrix I_p is shown in *Figure 2*. A Sub-PE (SPEi) is utilized for the vector-matrix multiplication of input vector I_{pi} and the weight matrix W_{ij}, where 'i' ranges from 1 to 'n' (corresponding to each sub-PE) and 'j' ranges from 1 to 8 (corresponding to each PE), resulting in an output vector of size 16 (represented as O_{p1}-O_{pn}). A vector called 'sum' of size 16 stores the intermediate values obtained after multiplying and accumulating the input vector (I_i) and the weight matrix (W_{ij}) in the individual sub-PEs.

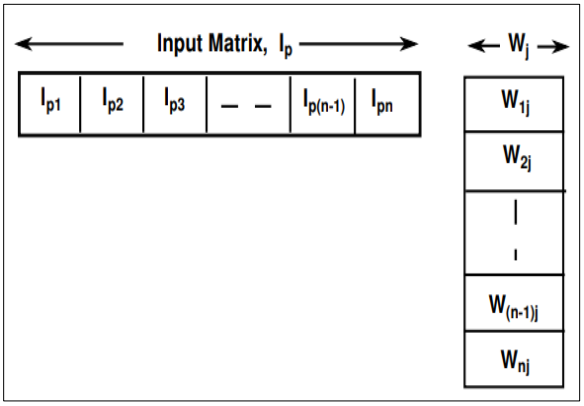

Figure 2 Division of input and weight across the sub-PEs

A tree adder is then used for the final computation of all the 16 ×1 output vectors from the sub-PEs to get the final output of size 16 ×1 from a PE. The structure of a PE with the division of column partitioned weight matrix W_j, input layer matrix I_p,

sub-PEs, and tree adder is shown, in *Figure 3*. Eight such outputs of size 16 ×1 are available from the execution of the eight PEs, which are then concatenated to get the final output of size 128 ×1 as shown in *Figure 4*. The naming convention for the designs implemented using this methodology follows a format like *Wtn1_BFn2* indicating a design with a weight level parallelism of 'n_1' (Number of sub-PEs in a PE are 'n_1') and a block factor of 'n_2' ('*sum*' array in the sub-PE is partitioned into 'n_2' blocks). In design-1 (*Wt8_BF2*), 8 sub-PEs are instantiated in a PE. The weights inputted into the PE, W_j, are again partitioned into 8 equal portions (W_{1j}-W_{8j}) and stored in separate block random access memory (BRAM). Similarly, I_p is also partitioned into I_{p1}-I_{p8} and stored in separate BRAM which are then simultaneously fed into each sub-PE.

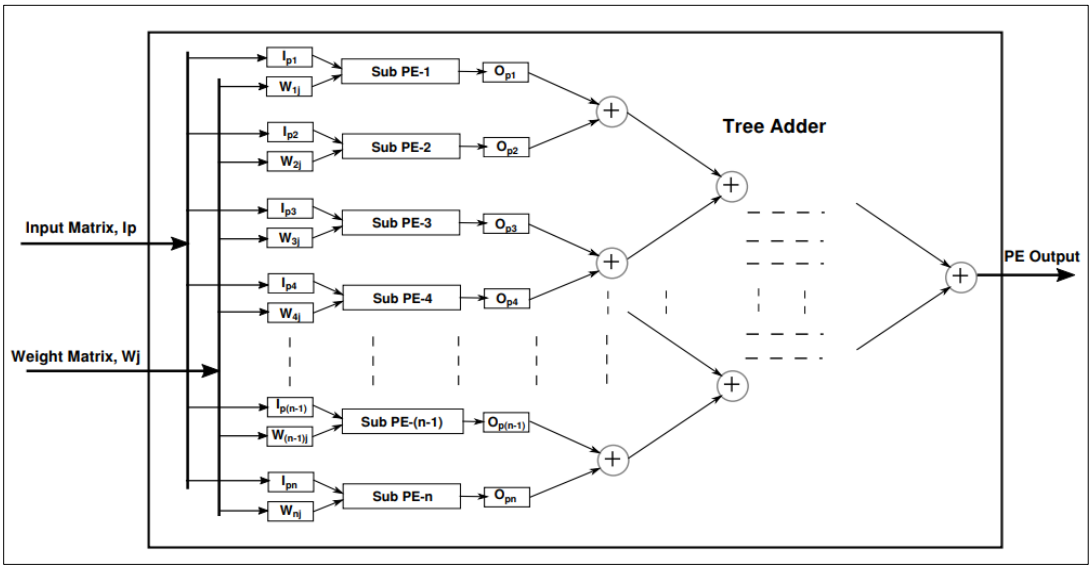

Figure 3 Structural diagram of a PE with n sub-Pes

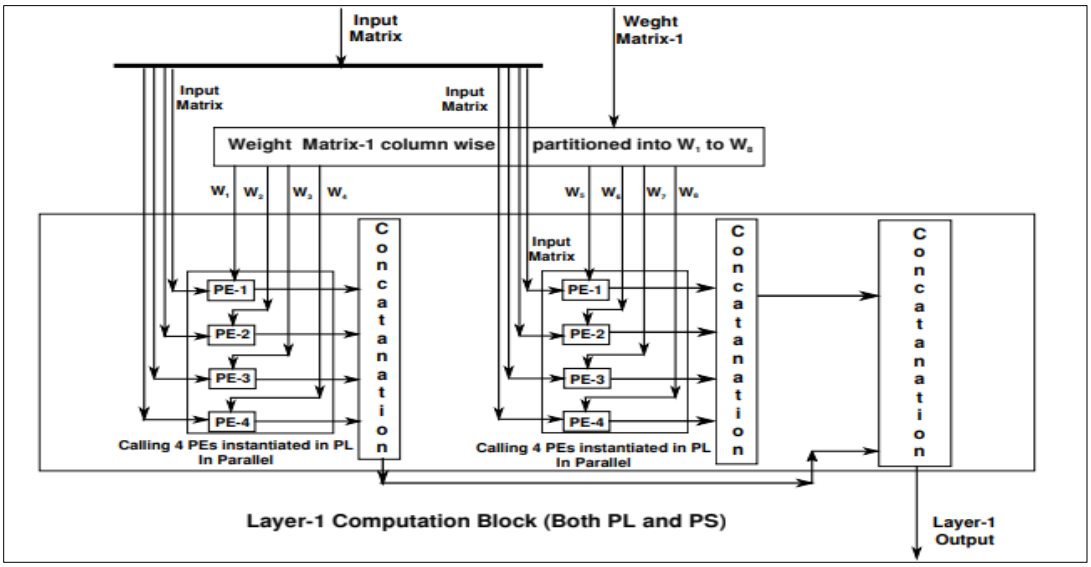

Figure 4 Structural diagram of (*Wt2_BF4*) with four PEs instantiated in parallel

Each BRAM is instantiated as a two-port device with two elements that can be accessed in parallel. Since the vector 'sum' needs to be accessed concurrently for the parallel computation of these operations, a pragma called *array_partitioning* [43] with a block factor of 2 has been used in this design. However,

partitioning done in this manner increases BRAM utilization. Another characteristic of this design is that only two PEs can be instantiated in the PL for parallel node-level computation due to resource utilization constraints.

The second design, *Wt8_BF4*, is similar to the first, with the difference being that a block factor of 4 is used for partitioning the vector sum, instead of 2. The increase in the block factor also results in a decrease in latency, since more values can be accessed in parallel compared to *Wt8_BF2*.

In the third design (*Wt4_BF4*), only 4 sub-PEs are instantiated in the PE, while the array partitioning for the sum array remains the same as in *Wt4_BF4*, i.e., a block factor of 4. The resources used for each sub-PE remain the same as in the previous designs, but with fewer sub-PEs in parallel, the resource utilization of a PE reduces, thus making it possible to increase the number of PEs in parallel compared to *Wt8_BF2* and *Wt8_BF4*. Thus, three PEs could be instantiated in the PL and could be called in parallel for the overall layer-1 computation. Since the PEs need to be called 8 times for the complete computation of layer-1, the overall computation can be done in 3 sequential operations, compared to 4 in the other two designs.

The fourth design, called *Wt2_BF4*, is similar to *Wt4_BF4*, except that the number of sub-PEs in the PE is reduced to 2 instead of 4. This reduction further decreases resource utilization compared to the previous designs. Consequently, it allows for an increase in the number of parallel PEs in the PL to four, as depicted in *Figure 4*, thereby reducing the number of sequential operations required for the overall computation of layer-1 to just two.

3.2.2Designs implemented using methodology-2

Two different designs are implemented using this methodology by varying the number ('n') of sub-PEs, to analyze the performance corresponding to various degrees of node-level parallelism. The performance corresponding to the weight level parallelism is examined by varying the number of parallel PEs in the PL. In these designs, the total computation is distributed amongst eight different PEs. In this methodology, the input layer vector (represented as I_p) of size 784 is partitioned into 8 equal parts (represented as I_{p1}-I_{p8}) of size 98 to make it individually available for the 8 PEs to exploit the weight level parallelism in the ANN. Similarly, the entire layer-1 weight matrix of dimension 784×128 is row partitioned into 8 equal parts of dimensions 98 ×128 (represented as W_1-W_8). Each PE takes a row-partitioned weight matrix W_i and the partitioned input layer matrix I_{pi} as the input.

In each PE, the partitioned weight matrix W_i is further partitioned column wise into W_{i1}-W_{in} (value of *'n'* is the same as the number of sub-PEs instantiated within the PE) which is then fed into the sub-PEs along with the partitioned input matrix (I_{pi}) for vector-matrix multiplication. Here, the number *'n'* also represents the node-level parallelism exploited in the layer-1. The outputs corresponding to each sub-PE (O_{p1}-O_{pn}), of size 128/*n*, are then concatenated to get the PE output of size 128. In this method, the output from the eight PEs is then added together to get the final layer-1 output. The partition of the input matrix, the layer-1 weight matrix and the PE structure are shown in *Figure 5*.

The naming convention for the designs implemented using this methodology follows the format *Ndn1_BFn2*, indicating a design with a node-level parallelism of n_1 (the number of sub-PEs in a PE is 'n_1') and a block factor of 'n_2' (the *'sum'* array in the sub-PE is partitioned into 'n_2' blocks). In the first design, called *Nd8_BF4*, eight sub-PEs are instantiated in the PE and two PEs in the PL for the layer-1 computation. The second design (*Nd2_BF4)*, is similar to the first, with the exception that the number of sub-PEs is two instead of eight. Resource utilization is reduced compared to the previous design, leading to the instantiation of four PEs in the PL.

3.3Implementation using HLS

The FPGA used in this paper is the PYNQ board, an SoC-FPGA, that has both PL and processing system (PS) [44]. As illustrated in *Figure 6*, the PS comprises double data rate synchronous dynamic random access memory (DDR DRAM), responsible for storing crucial components such as the input matrix, weight matrices, Bias, and Norm parameters necessary for computations. In the PL, PEs are instantiated specifically for layer-1 computations, while the remaining processing tasks are executed within the PS. The instantiation count of PEs varies depending on the specific design under consideration. Furthermore, the dimensions of the input and weight matrices, along with the desired size, also vary across different designs. Consequently, a control block is integrated into the PS, tailored to accommodate the current dataset dimensions (784×128 weight matrix and a 1×784 input matrix).

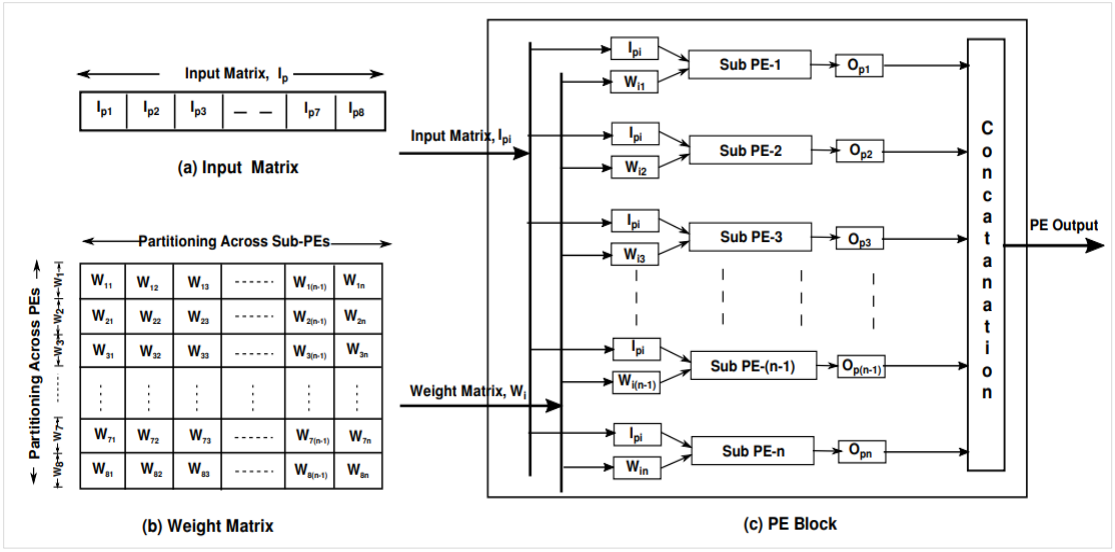

Figure 5 (a) Partition of input matrix **(b)** Partition of weight matrix **(c)** PE block structure

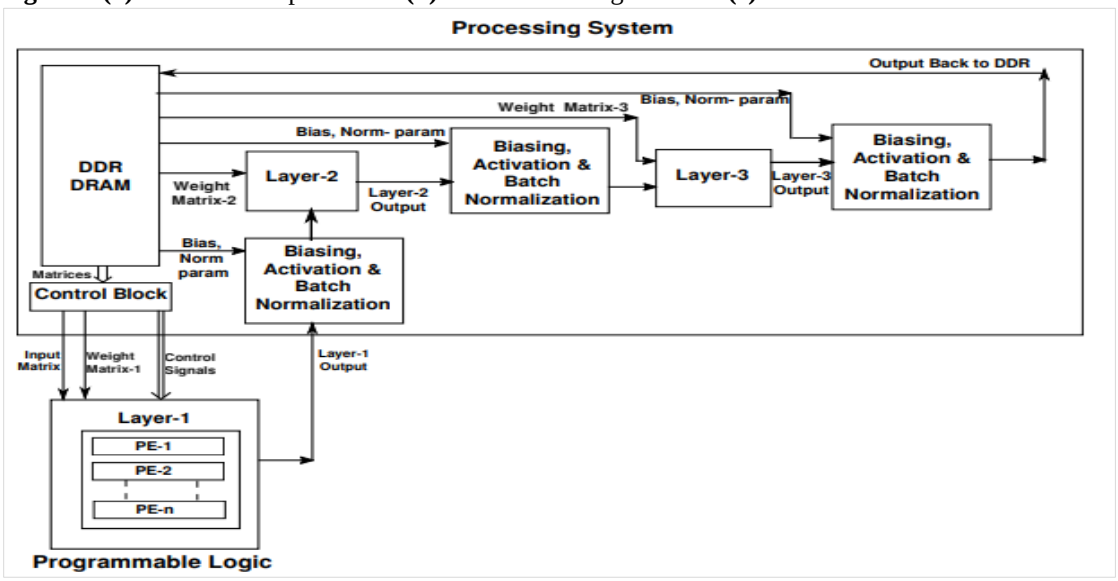

Figure 6 Distribution of computation between the PL and PS

This control block facilitates the selection and transfer of required memory segments to the PL, in addition to managing associated control signals. Through a flexible redesign of this control block, any proposed designs can be seamlessly adapted to meet the requirements of diverse ANN applications. The output from layer-1 is routed back to the PS to undergo subsequent layer computations. As shown in *Figure 6*, processes such as biasing, activation, and batch normalization are applied to the layer-1 output before being forwarded to layer-2. Within layer-2, computations utilize weight matrix-2 retrieved from the DRAM alongside the normalized data, with the resulting output undergoing further normalization incorporating bias and norm parameters from the DRAM. This processed data is then utilized in layer-3 computations, combined with weight matrix-3 from the DRAM. After additional normalization, the final output is stored in the DRAM. A high performance (HP) port is used for interfacing between the PL and the PS, but since it has cache coherence issues, the cache needs to be disabled before being called by the PE [45, 46].

The program corresponding to one PE was written in 'C' and converted to a customized digital IP using Vivado HLS. It was then instantiated multiple times and integrated with ZynQ Processor using Vivado

HLx. During the IP creation, the ports of the PE were declared as *m_axi* while in Vivado HLx the ports were customized as HP ports. The PE in the PL, with the help of these ports, can then access the input vector and weight matrices stored in the DDR-DRAM of the PS using pointer-based memory-mapped instructions.

As mentioned earlier, in methodology-1, the 784 ×16 weight matrix in a PE is row partitioned into *'n'* arrays of size (784/*n*) ×16, with each array being utilized in a sub-PE. These arrays are copied from the DDR-DRAM of the PS to the BRAMs using the *memcpy* command. The value of *'n'* varies between the designs and indicates the number of sub-PEs in a PE. Each sub-PE performs a vector-matrix multiplication between a vector of size (784/n) and a matrix of size (784/*n*) ×16. *Unroll* and *pipeline* pragmas are used in each sub-PE to extract maximum parallelism and to reduce the latency of execution [43]. The pragma *unroll* is used to replicate the body of a loop, which otherwise would be done sequentially, to achieve shorter latency and higher throughput. The pragma *pipeline* is used to instruct the pipelining of the execution [43]. The 16 ×1 output obtained from each of the *'n'* sub-PEs is then added together by constructing a tree adder with the pragma *pipeline*, to get a 16×1 vector as the output of the PE.

Similar strategies are applied in methodology-2, however, the output from the PE is determined by concatenating the output from each sub-PE instead of adding using a tree adder. The size of the input vector and weight matrix for PE is 98 and 98 ×128 respectively, while that for the sub-PE is 98 and 98 ×(128/*n*). In this method also, the pragmas *unroll*, *pipeline*, and *array_partitioning* are used for the vector-matrix multiplication operation in the sub-PE.

Every sub-PE utilizes BRAMs to store the input and output arrays. Each BRAM has a dual port configuration, meaning that only 2 elements of an array can be accessed at a time. To access more than two elements in a BRAM, the pragma *array_partitioning* can be used [43], however, it may increase the BRAM utilization for the array mentioned.

To reduce the wastage of space in BRAM the pragma *array_mapping* [43] is used for the implementation. It will combine the mentioned arrays and may decrease the BRAM requirement however; the parallel accessibility will be reduced. Once the size of BRAMs is available and the size of arrays needed, the pragma *array_partitioning* or *array_mapping* can be used for the optimization.

Once timing constraints are specified and the `*C/C++*'code for a PE is synthesized in Vivado HLS, it is converted into RTL. An IP can then be created once the synthesis is complete, ensuring timing constraints for resource utilization, parallelism, and latency are all met [45, 46]. One can utilize the IPs available in Vivado HLx such as Zynq processor, and AXI-interconnect to integrate with our custom digital IP [45, 46]. Since the PL and the Zynq PS need to be combined, the frequency and the type of port used for the Zynq Processor should be customized to match that of the custom IP.

Figure 7 shows the block designed in Vivado HLx for the design *Wt2_BF4* by integrating Zynq PS with four copies of the corresponding custom-designed IPs. The connections within them and the assignment of addresses are all automated by the software. Once a top-level HDL wrapper is created, it is synthesized and implemented, which is when the fitting and routing on the FPGA are done. This process is followed by static timing analysis, after which a bit stream is generated. The complete system is then exported into Xilinx SDK.

In the SDK, a program in *'C'* is developed for the processor, which includes all the computation needed to be done by the processor and the communication required between the PL and the PS [45, 46]. The PS has DDR-DRAM, which stores all the weights/parameters generated after training the ANN in Python. A control block is also designed in the PS for the selection and transfer of suitable parts of the memory to the PL as and when necessary and for storing the corresponding data back to the DDR-DRAM. The control signals for the functioning of all the PEs in the PL are also generated by the control block. After the code is created, the SDK is built, and an elf file is generated. The results of the FPGA corresponding to the latency and its accuracy are then analyzed after it is loaded with the elf file and the generated bit stream.

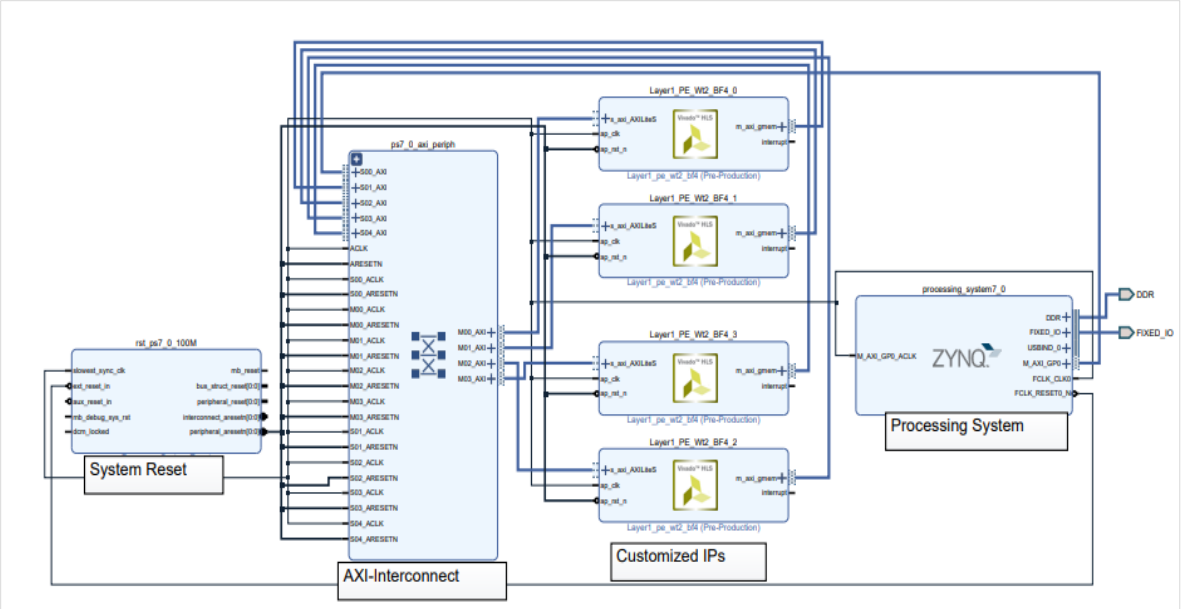

Figure 7 Vivado HLx block diagram with PL & PS

4.Results

The ANN model is implemented and analyzed using a Xilinx SoC-based PYNQ Board [44] which integrates hard dual-core ARM Cortex-A9, 666 MHz PS with the PL. The PL contains 4.9Mb BRAM, 220 DSP slices, etc [47]. The software tools used for the analysis are Xilinx's Vivado HLS, Vivado HLx, and SDK. As discussed, the custom-designed IP corresponding to the PE for layer-1 computation is created using Vivado HLS. Multiple IPs are instantiated in the PL to extract the available node-level parallelism. The remaining parts of the computation are performed in the ARM processor. The resource utilization and Latency per PE corresponding to the four designs using methodology-1 observed during the execution on the FPGA are shown in *Table 1*.

Table 1 clearly illustrates a trend wherein weight level parallelism increases as one progresses from serial number 1-3. This increment is reflected in the instantiation of more sub-PEs within the PE, resulting in a notable increase in DSP, LUT, and FF resource utilization percentages. Additionally, it is evident that increasing weight level parallelism from serial number 1 to 3 in *Table 1* results in reduced per PE computational latency. Furthermore, a comparison of the last two rows of *Table 1* reveals that reducing the block factor leads to a considerable increase in computational latency. The total latency corresponding to the complete ANN computation depends upon the latency for each PE and the number of PEs instantiated in the PL. The performance observed in each design for the complete ANN computation is shown in *Table 2*. Columns 2 to 6 indicate the number of PEs instantiated for the complete ANN implementation, the total computational latency with multiple PEs in the PL and other computations in the ARM processor, the latency when all the computations are in the ARM processor, the acceleration achieved using both PL and PS for the ANN computation and the accuracy, respectively.

Table 1 Percentage resource utilisation and latency for various designs per PE with methodology-1

Serial Number	Designs	BRAM	DSP	LUT	FF	Per PE Latency (µs)
1	*Wt2_BF4*	13	10	15	10	1058
2	*Wt4_BF4*	14	21	24	17	930
3	*Wt8_BF4*	15	43	42	31	833
4	*Wt8_BF2*	27	50	39	22	1075

Table 2 Performance observed with multiple PEs in PL using method-1

Designs	No of PEs in PL	Overall Latency with PL &PS (µs)	Latency with only PS (ARM) (µs)	Acceleration	%Accuracy
Wt2_BF4	4	2721	6497	2.39	97.2
Wt4_BF4	3	3315	6497	1.96	97.2
Wt8_BF4	2	3938	6497	1.65	97.2
Wt8_BF2	2	4913	6497	1.32	97.2

Column 2 shows that when the weight level parallelism is less utilized (a smaller number of sub-PEs in a PE, as in the case of *Wt2_BF4)* a larger number of PEs could be instantiated in the PL. Thus, more node-level parallelism could be exploited as explained in section 3.2.1. This leads to less overall latency, as shown in column-3, even though per PE latency is more, as demonstrated in *Figure 8.*

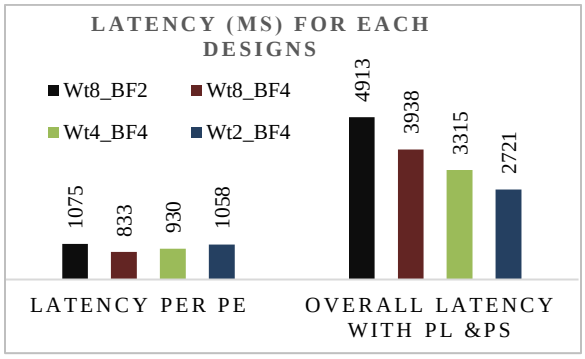

Figure 8 Latency per PE and total latency with PL and PS for each design in methodology-1

The performance analysis in *Table 2* can be summarized in relation to the block factor and sub-PEs as follows: Increasing the block factor reduces latency by allowing parallel access to a larger range of values. At the same time, lowering the number of sub-PEs operating in parallel decreases the resource demand per PE. This reduction allows for more parallel PEs within the PL. The greater number of parallel PEs reduces the need for sequential operations during layer-1 computation, ultimately lowering the overall computational latency.

When all the computations, including the layer-1 computation shown in *Figure 6*, are performed in the ARM processor of 666 MHz, the total latency is observed as 6497 µs and is mentioned in column-4 of *Table 2*. The acceleration achieved for each design by performing the layer-1 computation in PL is calculated as the ratio of values in column-4 to column-3 and is shown in column-5. Thus, with the *Wt2_BF4* design, the total latency for the ANN computation (with layer-1 computation in PL) is 2721 µs, which is 2.39 faster compared to all computations in the ARM processor.

The last column of *Table 2* shows the accuracy of ANN computation which remains consistent with that of the Python implementation, as single-precision floating-point computation is utilized across all designs, including computations in the PL. While this choice results in increased resource utilization and latency compared to fixed-point implementation, it ensures accuracy equivalent to that of the Python implementation. Consequently, our designs achieve a notable acceleration while maintaining a high accuracy rate of 97.2%.

Total resource utilization for each design can be computed by multiplying the number of PEs in PL (second column in *Table 2*) with the corresponding per PE resource utilization given in *Table 1*. For example, in the case of the *Wt2_BF4* design, total utilization becomes 4 times the resource utilization given in the last row in *Table 1*. The analysis reveals that resource utilization is highest with Wt8_BF2 and lowest with *Wt2_BF4*, both operating at the same clock frequency of 150 MHz. Consequently, the corresponding power requirements vary, ranging from 4.382 W for *Wt8_BF2* to 3.234 W for *Wt2_BF4.* The resource utilization and the computational latency per PE corresponding to the designs using method-2 are shown in *Table 3. Table 4* presents performance metrics for different designs utilizing multiple PEs in the PL with method-2. As in *Table 2*, an increase in sub-PEs results in higher resource utilization, which restricts the number of PEs that can be instantiated in the PL. Consequently, this results in increased computational latency, exemplified by *Nd8_BF4*, despite exhibiting lower latency per PE as shown in *Figure 9.* The total latency observed in the case of *Nd2_BF4* is only 2592µs, which is the lowest compared to all designs. It leads to the acceleration of 2.5X with *Nd2_BF4* design, while computing using both PL and PS instead of using only the ARM processor.

Table 3 Percentage resource utilization and latency for various designs per PE using method-2

Serial Number	Designs	BRAM	DSP	LUT	FF	per PE Latency (µs)
1	*Nd2_BF4*	20	21	16	11	904
2	*Nd8_BF4*	16	37	39	29	728

Table 4 Performance analysis for various designs with multiple PEs in PL using method-2

Designs	No of PEs in PL	Latency with PL & PS (µs)	Latency with only PS (ARM) (µs)	Acceleration	%Accuracy
Nd2_BF4	4	2592	6497	2.5	97.2
Nd8_BF4	2	3594	6497	1.8	97.2

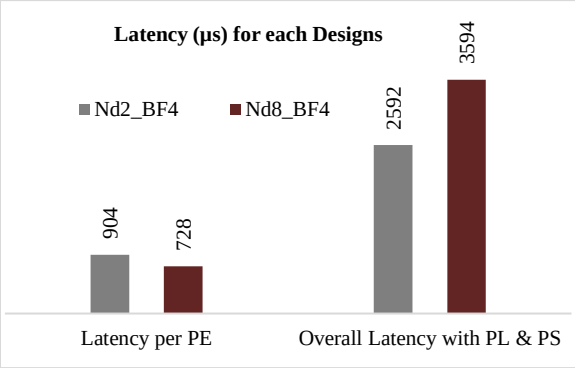

Figure 9 Latency per PE and overall latency with PL and PS for each design in methodology-2

As previously mentioned, the total resource utilization for each design can be calculated by multiplying the number of PEs in the PL (shown in the second column of *Table 4*) by the per-PE resource utilization listed in *Table 3*. For example, in the case of *Nd8_BF4*, the total utilization is twice the resources indicated in the last row of *Table 3*.

In Methodology-2, it is notable that LUT and FF utilization are higher in the *Nd8_BF4* configuration, while BRAM and DSP utilization are greater in *Nd2_BF4*, with both operating at the same clock frequency of 150 MHz. Consequently, the power requirements for both configurations are similar, falling within the range of 3W to 4W.

5.Discussion

Analysis of *Tables 1* and *3* reveals that increasing the number of sub-PEs per PE (from *Wt2_BF4* to *Wt8_BF4* and from *Nd2_BF4* to *Nd8_BF4*) significantly improves both weight-level and node-level parallelism, thereby reducing computational latency per PE. However, this performance gain comes with increased resource utilization. Raising the block factor reduces computational latency per PE but can lead to higher LUT utilization and lower BRAM usage, as seen in the comparison between *Wt8_BF2* and *Wt8_BF4* in *Table 1*. This occurs because, after partitioning arrays to accommodate a larger block factor, smaller arrays are more likely to be implemented using LUTs rather than BRAM. *Tables 2* and *4* demonstrate that increasing the number of instantiated PEs in the PL enables greater parallel processing, resulting in a reduction in overall latency.

However, as shown in *Table 5*, while increasing parallelism within each PE reduces per PE computational latency, it also raises resource utilization, which limits the number of PEs that can be instantiated simultaneously. For example, *Wt8_BF4* with method-1 achieves lower per PE latency, but due to higher resource consumption, only two PEs can be instantiated in parallel, leading to higher overall latency compared to *Wt2_BF4*. A similar pattern is observed for *Nd8_BF4* versus *Nd2_BF4*. This analysis suggests that, despite slightly higher per PE computational latency, instantiating more PEs in the PL can lead to lower overall latency.

In method-1, the advantage of adding more PEs is limited by the sequential access of the input matrix from DDR-DRAM to BRAMs, which increases communication latency. On the other hand, method-2 allows for concurrent communication and computation by enabling the parallel instantiation of PEs, significantly reducing total latency, as highlighted in *Table 5*. This concurrent approach in method-2 results in better overall acceleration compared to method-1. The overall analysis indicates the importance of balancing per-PE computational latency with the number of PEs instantiated in the PL. While exploiting parallelism within each PE reduces per PE computational latency, it also increases resource usage, which limits the extent to which PEs can be instantiated in parallel.

Table 5 Performance analysis for various designs with multiple PEs in PL using both method-1 and 2

Serial Number	Designs	Method	per PE computational Latency (µs)	No of PEs instantiated in parallel in PL	Latency with PL & PS (µs)	Latency with only PS (ARM) (µs)	Acceleration
1	*Wt2_BF4*	method-1	1058	4	2721	6497	2.39
2	*Wt8_BF4*	method-1	833	2	3938	6497	1.65
3	*Nd2_BF4*	method-2	904	4	2592	6497	2.5
4	*Nd8_BF4*	method-2	728	2	3594	6497	1.8

5.1Limitation and the possible solution

The limited resources available on the PYNQ board pose a constraint on acceleration capabilities. However, the strategy outlined in this paper is designed to be scalable and adaptable for various ANN applications, by redesigning the control blocks. Even with the constrained resources of the PYNQ board, the implementation surpasses ARM processors significantly. With access to more resources or through deployment across multiple FPGAs, its acceleration performance is anticipated to rival that of GPUs and many conventional processors, emphasizing its potential for significant speedup enhancements.

A complete list of abbreviations is listed in *Appendix I.*

6.Conclusion and future work

This paper presents various methods for implementing ANN computations on an FPGA for prototyping and acceleration. While the classification of handwritten digits is used as a case study, the proposed approach is highly adaptable to any ANN application, irrespective of the number of neurons per layer or the input size. The methods focus on executing computations for the largest layer of the ANN within the PL, effectively leveraging available node-level and weight-level parallelism. This is accomplished by designing a 150 MHz digital system with a PE composed of multiple sub-PEs, utilizing Vivado tools. Multiple PEs are instantiated within the PL, while a control block in the PS is designed to manage the selection of PEs, their inputs, and control signals.

The performance of the two methodologies was thoroughly analyzed, demonstrating a significant acceleration of 2.5 times with a high accuracy rate of 97.2% (using single-precision floating-point computations), compared to executing all computations in the PS running at 666 MHz. This approach is scalable, and the flexibility of Vivado tools make it adaptable for various applications. The results indicate that, even with limited resources, the PYNQ FPGA implementation surpasses ARM performance, and utilizing multiple FPGAs (or a single FPGA with enhanced resources), could yield performance levels comparable to those of general-purpose processors. Moreover, the FPGA prototype design can be extended for developing high-performance application specific integrated circuits (ASICs) tailored for ANN applications.

Acknowledgment

None.

Conflicts of interest

The authors have no conflicts of interest to declare.

Data availability

The MNIST dataset utilized in this study is publicly accessible and can be found at http: //yann.lecun.com/exdb/mnist/

Author's contribution statement

Both authors contributed to the study's conception, algorithm development, and analysis. Additionally, they both drafted the manuscript, reviewed it, and approved the final version.

References

[1] Sarker IH. Machine learning: algorithms, real-world applications and research directions. SN Computer Science. 2021; 2(3):160.

[2] Mahesh B. Machine learning algorithms-a review. International Journal of Science and Research. 2020; 9(1):381-6.

[3] Hall W, Tian Y. Neural networks training on graphics processing unit (GPU) using dynamic parallelism (DP). In proceedings of SAI intelligent systems conference 2022 (pp. 811-8). Cham: Springer International Publishing.

[4] Jayanthi B, Kumar LS. Implementation of neural networks in FPGA. Indian Journal of Radio & Space Physics. 2021; 50(2).

[5] Luk W. Heterogeneous reconfigurable accelerators: trends and perspectives. In 60th ACM/IEEE design automation conference 2023 (pp. 1-2). IEEE.

[6] Boutros A, Nurvitadhi E, Ma R, Gribok S, Zhao Z, Hoe JC, et al. Beyond peak performance: comparing the real performance of AI-optimized FPGAs and

GPUs. In international conference on field-programmable technology 2020 (pp. 10-9). IEEE.
[7] Hu Y, Liu Y, Liu Z. A survey on convolutional neural network accelerators: GPU, FPGA and ASIC. In 14th international conference on computer research and development 2022 (pp. 100-7). IEEE.
[8] Liu C. Yolov2 acceleration using embedded GPU and FPGAS: pros, cons, and a hybrid method. Evolutionary Intelligence. 2022; 15(4):2581-7.
[9] Boutros A, Betz V. FPGA architecture: principles and progression. IEEE Circuits and Systems Magazine. 2021; 21(2):4-29.
[10] Ersoy M, Kumral CD. Realization of artificial neural networks on FPGA. In artificial intelligence and applied mathematics in engineering problems: proceedings of the international conference on artificial intelligence and applied mathematics in engineering 2020 (pp. 418-28). Springer International Publishing.
[11] Ney J, Hammoud B, Dörner S, Herrmann M, Clausius J, Ten BS, et al. Efficient FPGA implementation of an ANN-based demapper using cross-layer analysis. Electronics. 2022; 11(7):1-22.
[12] Atibi M, Boussaa M, Bennis A, Atouf I. Real-time implementation of artificial neural network in FPGA platform. In embedded systems and artificial intelligence: proceedings of ESAI, Fez, Morocco 2020 (pp. 3-13). Springer Singapore.
[13] Vineetha KV, Reddy MM, Ramesh C, Kurup DG. An efficient design methodology to speed up the FPGA implementation of artificial neural networks. Engineering Science and Technology, an International Journal. 2023; 47:101542.
[14] Borhani A, Goharinejad MH, Zarandi HR. FAST: FPGA acceleration of neural networks training. In 12th international conference on computer and knowledge engineering 2022 (pp. 492-7). IEEE.
[15] https://fpgainsights.com/category/FPGA/. Accessed 24 August 2024.
[16] Jia X, Zhang Y, Liu G, Yang X, Zhang T, Zheng J, et al. XVDPU: a high-performance CNN accelerator on the versal platform powered by the AI engine. ACM Transactions on Reconfigurable Technology and Systems. 2024; 17(2):1-24.
[17] Carpegna A, Savino A, Di CS. Spiker: an FPGA-optimized hardware accelerator for spiking neural networks. In computer society annual symposium on VLSI 2022 (pp. 14-9). IEEE.
[18] Pham-quoc C, Nguyen XQ, Thinh TN. Towards an FPGA-targeted hardware/software co-design framework for CNN-based edge computing. Mobile Networks and Applications. 2022; 27(5):2024-35.
[19] Yawalkar PM, Kharat MU. Automatic handwritten character recognition of Devanagari language: a hybrid training algorithm for neural network. Evolutionary Intelligence. 2022; 15(2):1499-516.
[20] Al-rikabi H, Renczes B. Floating-point quantization analysis of multi-layer perceptron artificial neural networks. Journal of Signal Processing Systems. 2024:1-12.
[21] Ortega-zamorano F, Jerez JM, Gómez I, Franco L. Layer multiplexing FPGA implementation for deep back-propagation learning. Integrated Computer-Aided Engineering. 2017; 24(2):171-85.
[22] Saady MM, Essai MH. Hardware implementation of neural network-based engine model using FPGA. Alexandria Engineering Journal. 2022; 61(12):12039-50.
[23] Xie X, Wu C. WPU: a FPGA-based scalable, efficient and software/hardware co-design deep neural network inference acceleration processor. In international conference on high performance big data and intelligent systems 2021 (pp. 1-5). IEEE.
[24] Elsaid K, Safar M, El-kharashi MW. Optimized FPGA architecture for machine learning applications using posit multipliers. In international conference on microelectronics 2022 (pp. 50-3). IEEE.
[25] Elsaid K, El-kharashi MW, Safar M. An optimized FPGA architecture for machine learning applications. AEU-International Journal of Electronics and Communications. 2024; 174:155011.
[26] Nobari M, Jahanirad H. FPGA-based implementation of deep neural network using stochastic computing. Applied Soft Computing. 2023; 137:110166.
[27] Lingala SS, Bedekar S, Tyagi P, Saha P, Shahane P. FPGA based implementation of neural network. In international conference on advances in computing, communication and applied informatics 2022 (pp. 1-5). IEEE.
[28] Gdaim S, Mtibaa A, Mimouni MF. Artificial neural network-based DTC of an induction machine with experimental implementation on FPGA. Engineering Applications of Artificial Intelligence. 2023; 121:105972.
[29] Kim H. Review of optimal convolutional neural network accelerator platforms for mobile devices. Journal of Computing Science and Engineering. 2022; 16(2):113-9.
[30] Hong H, Choi D, Kim N, Lee H, Kang B, Kang H, et al. Survey of convolutional neural network accelerators on field-programmable gate array platforms: architectures and optimization techniques. Journal of Real-Time Image Processing. 2024; 21(3):64.
[31] Kumbhar RR, Radhika P, Mane D. Design and optimization of an on-chip artificial neural network on FPGA for recognizing handwritten digits. In international conference on recent advances in electrical, electronics, ubiquitous communication, and computational intelligence 2023 (pp. 1-5). IEEE.
[32] Sowmya N, Kumar J, Biswal PK, Roy S, Pradhan S. Neuromorphic processor design and FPGA implementation for handwritten digits employing spiking neural network. International Journal of Computing and Digital Systems. 2023; 14(1):679-89.
[33] Khan MS, Yadav P, Verma R, Sreedevi I. FPGA simulation of fingertip digit recognition using CNN. In 7th international conference on signal processing and integrated networks 2020 (pp. 1072-7). IEEE.

[34] Otter DW, Medina JR, Kalita JK. A survey of the usages of deep learning for natural language processing. IEEE Transactions on Neural Networks and Learning Systems. 2020; 32(2):604-24.
[35] Pramodhini R, Harakannanavar SS, Akshay CN, Rakshith N, Shrivastava R, Gupta A. Robust handwritten digit recognition system using hybrid artificial neural network on FPGA. In 2nd Mysore sub section international conference 2022 (pp. 1-5). IEEE.
[36] Shen G, Li J, Zhou Z, Chen X. FPGA-based neural network acceleration for handwritten digit recognition. In international conference on internet of things as a service 2020 (pp. 182-95). Cham: Springer International Publishing.
[37] Mittal H, Sharma A, Perumal T. FPGA implementation of handwritten number recognition using artificial neural network. In 8th global conference on consumer electronics 2019 (pp. 1010-1). IEEE.
[38] Cong J, Lau J, Liu G, Neuendorffer S, Pan P, Vissers K, et al. FPGA HLS today: successes, challenges, and opportunities. ACM Transactions on Reconfigurable Technology and Systems. 2022; 15(4):1-42.
[39] Abdelhamid RB, Kuwazawa G, Yamaguchi Y. Quantitative study of floating-point precision on modern FPGAs. In proceedings of the 13th international symposium on highly efficient accelerators and reconfigurable technologies 2023 (pp. 49-58). ACM.
[40] Keilbart C, Gao Y, Chua M, Matthews E, Wilton SJ, Shannon L. Designing an IEEE-compliant FPU that supports configurable precision for soft processors. ACM Transactions on Reconfigurable Technology and Systems. 2024; 17(2):1-32.
[41] https://www.tensorflow.org/datasets/catalog/mnist. Accessed 24 August 2024.
[42] https://www.kaggle.com/code/benroshan/digit-fashion-mnist-ann/notebook. Accessed 24 August 2024.
[43] https://docs.amd.com/v/u/en-US/ug902-vivado-high-level-synthesis. Accessed 24 August 2024.
[44] https://www.xilinx.com/support/university/xup-boards/XUPPYNQ-Z2.html. Accessed 24 August 2024.
[45] Namboothiripad MK, Datar MJ, Chandorkar MC, Patkar SB. Accelerator for real-time emulation of modular-multilevel-converter using FPGA. In 21st workshop on control and modeling for power electronics 2020 (pp. 1-7). IEEE.
[46] Namboothiripad MK, Datar MJ, Chandorkar MC, Patkar SB. FPGA accelerator for real-time emulation of power electronic systems using multiport decomposition. IEEE Transactions on Industry Applications. 2020; 56(6):6674-86.
[47] https://www.avnet.com/wps/portal/apac/products/c/xilinx-pynq. Accessed 24 August 2024.

Mini K. Namboothiripad received her B.Tech. degree in Electrical and Electronics Engineering from the Government Engineering College Thrissur, University of Calicut, Kerala, India, in 1995. She obtained her M.Tech. degree in 2011 and her Ph.D. degree in 2021, both in Electrical Engineering from the Indian Institute of Technology Bombay, Mumbai, India. Since 2001, she has been working as an Assistant Professor in the Department of Electrical Engineering at Fr. C. Rodrigues Institute of Technology, Navi Mumbai, India. Her research interests include FPGA-based reconfigurable computing, real-time simulation, mathematical modeling, and control of electrical systems.
Email: mini.n@fcrit.ac.in

Gayathri Vadhyan received her B.Tech. degree in Electronics and Instrumentation Engineering from Vellore Institute of Technology, Vellore, Tamil Nadu, India, in 2021. Her research interests include Hardware-Software Co-Design, Artificial Neural Networks, Machine Learning, Processor Design, and FPGA-based computing.
Email: gvadhyan@gmail.com

Appendix I

S. No.	Abbreviation	Description
1	ANN	Artificial Neural Network
2	ARM	Advanced Reduced Instruction Set Computer Machine
3	ASIC	Application Specific Integrated Circuit
4	BRAM	Block Random Access Memory
5	DDR-DRAM	Double Data Rate Synchronous Dynamic Random Access Memory
6	DSP	Digital Signal Processor
7	FF	Flip-Flop
8	FPGA	Field Programmable Gate Arrays
9	GPU	Graphical Processing Unit
10	HDL	Hardware Descriptive Language
11	HLS	High Level Synthesis
12	HP	High Performance
13	I/O	Input/Output
14	IP	Intellectual Property
15	LUT	Look Up Table
16	MHz	Mega Hertz
17	MNIST	Modified National Institute of Standards and Technology Database
18	PE	Processing Elements
19	PL	Programmable Logic
20	PS	Processing System
21	PYNQ	Python based FPGA
22	ReLU	Rectified Linear Unit
23	RTL	Register Transfer Level
24	SoC	System on Chip