electronics

MDPI

Article

FPQNet: Fully Pipelined and Quantized CNN for Ultra-Low Latency Image Classification on FPGAs Using OpenCAPI

Mengfei Ji [1,2,*], Zaid Al-Ars [2,*], Peter Hofstee [2], Yuchun Chang [3] and Baolin Zhang [1]

1 State Key Laboratory on Integrated Optoelectronics, College of Electronic Science & Engineering, Jilin University, Changchun 130012, China
2 The Department of Quantum & Computer Engineering, Delft University of Technology, 2628 CD Delft, The Netherlands
3 School of Microelectronics, Dalian University of Technology, Dalian 116620, China
* Correspondence: jimengfei007@outlook.com (M.J.); z.al-ars@tudelft.nl (Z.A.-A.)

Abstract: Convolutional neural networks (CNNs) are to be effective in many application domains, especially in the computer vision area. In order to achieve lower latency CNN processing, and reduce power consumption, developers are experimenting with using FPGAs to accelerate CNN processing in several applications. Current FPGA CNN accelerators usually use the same acceleration approaches as GPUs, where operations from different network layers are mapped to the same hardware units working in a multiplexed manner. This will result in high flexibility in implementing different types of CNNs; however, this will degrade the latency that accelerators can achieve. Alternatively, we can reduce the latency of the accelerator by pipelining the processing of consecutive layers, at the expense of more FPGA resources. The continued increase in hardware resources available in FPGAs makes such implementations feasible for latency-critical application domains. In this paper, we present FPQNet, a fully pipelined and quantized CNN FPGA implementation that is channel-parallel, layer-pipelined, and network-parallel, to decrease latency and increase throughput, combined with quantization methods to optimize hardware utilization. In addition, we optimize this hardware architecture for the HDMI timing standard to avoid extra hardware utilization. This makes it possible for the accelerator to handle video datasets. We present prototypes of the FPQNet CNN network implementations on an Alpha Data 9H7 FPGA, connected with an OpenCAPI interface, to demonstrate architecture capabilities. Results show that with a 250 MHz clock frequency, an optimized LeNet-5 design is able to achieve latencies as low as 9.32 µs with an accuracy of 98.8% on the MNIST dataset, making it feasible for utilization in high frame rate video processing applications. With 10 hardware kernels working concurrently, the throughput is as high as 1108 GOPs. The methods in this paper are suitable for many other CNNs. Our analysis shows that the latency of AlexNet, ZFNet, OverFeat-Fast, and OverFeat-Accurate can be as low as 69.27, 66.95, 182.98, and 132.6 µs, using the architecture introduced in this paper, respectively.

Keywords: CNNs; FPGA acceleration; HDMI; OpenCAPI; layer pipeline; channel parallelization

check for updates

Citation: Ji, M.; Al-Ars, Z.; Hofstee, P.; Chang, Y.; Zhang, B. FPQNet: Fully Pipelined and Quantized CNN for Ultra-Low Latency Image Classification on FPGAs Using OpenCAPI. *Electronics* **2023**, *12*, 4085. https://doi.org/10.3390/electronics12194085

Academic Editor: Alexander Barkalov

Received: 19 August 2023
Revised: 22 September 2023
Accepted: 25 September 2023
Published: 29 September 2023

1. Introduction

Convolutional neural networks (CNNs) have rapidly developed as effective tools for various computer vision tasks, such as image recognition [1], image classification [2,3], object detection [4,5], and image enhancement [6]. Instead of depending on manual feature engineering, as is the case in traditional machine learning algorithms, CNNs use feature-learning from large training sets; they leverage their superior learning capabilities to achieve accuracy that is comparable to, or even surpasses, human perception.

With the demands of low latency and low power consumption, CPU accelerators are not capable enough to meet this demand. Graphic processing units (GPUs) have been widely used for CNN inference by multiple researchers [7,8] because they can support multi-core parallel computing to achieve high throughput. However, the high power consumption of GPUs and their relatively high latency make them unsuitable for embedded, real-time applications. In contrast, FPGAs are good candidate accelerators in this space because of their large number of parallel computing resources, low latency, low power consumption, reprogrammable architectures, and high flexibility [9,10].

Current CNN FPGA accelerators are usually based on SIMD structures that are similar to GPUs [11,12]. Within this framework, different network layer operations are mapped to shared hardware units, allowing concurrent execution. This multiplexed execution of operations from diverse layers not only offers adaptability but also optimizes hardware resource utilization. However, they are unable to deliver the optimal latency or throughput performance of any particular neural network. The emergence of flow structure accelerators solved the problem [13,14]. Flow structures map all the network layers on-chip and encourage a dynamic interplay between layer parallelism and layer pipelining, effectively reducing the idle time associated with traditional processing. The flow structure reduces the latency of the networks and enhances the throughput. However, they require much more hardware resources, making them hard to implement on FPGAs. In this paper, we propose multiple quantization methods and model compression methods to reduce the sizes of the models, making the implementation of flow CNN accelerator structures more feasible. In addition, our optimized flow structure with the HDMI timing standard helps reduce the extra restoration of the intermediate feature maps, which helps alleviate hardware resource constraints.

In this paper, we present FPQNet, a fully pipelined and quantized CNN, which uses an optimized flow structure for the HDMI timing standard to implement a proof-of-concept CNN accelerator on FPGAs and mitigate the high hardware requirements by optimizing the amount of memory needed between different layers. We also implement ten convolutional neural network instances in the system to increase the throughput. In addition, we deploy several quantization methods to make their FPGA implementation more feasible. We integrate the design using an OpenCAPI interface to ensure high bandwidth communication with the host processor and prevent data communication bottlenecks. In the end, we also show the results of the models implemented by FINN [15], as well as the advantages and disadvantages of the implementation in this paper compared with FINN implementations.

The contributions of this paper are as follows:

1. We optimize the fully parallelized channel and fully pipelined layer structure with the HDMI timing standard to avoid extra data transfer and save hardware resources.
2. We integrate the accelerator with a high-bandwidth OpenCAPI interface.
3. We combine several proposed quantization and model compression methods to save hardware resources.

The rest of the paper is organized as follows. In Section 2, we provide the background knowledge used in this paper. In Section 3, we discuss the related research work of CNN accelerators. In Section 4, we present optimization methods to reduce the CNN hardware utilization. Section 5 focuses on the FPGA hardware structure of each component. Section 6 provides an evaluation of the performance that this design can achieve. Section 7 concludes this paper.

2. Background

In this section, we introduce the CNN structures, the HDMI timing standard, and the OpenCAPI interface, which are important technologies used in this paper.

2.1. CNN Structures

In this paper, we use the LeNet-5 model to implement in hardware. This model is widely used in many application scenarios, for example, medical diagnoses [16], signal processing [17], agriculture [18], etc. In addition, this model can be combined with the hardware flow structure and the full channel parallelism structure to achieve ultra-low latency, which is the focus of this paper. More complex models, such as those with branches like ResNet, are unable to achieve full channel parallelism, so they require longer latency times to process.

LeNet-5 is a basic and efficient convolutional neural network for handwritten digit recognition and other image recognition [19]. Figure 1 shows the structure of LeNet-5. Excluding the input layer, there are 7 layers, which are: C1 convolutional layer, S2 pooling layer, C3 convolutional layer, S4 pooling layer, C5 convolutional layer, F6 fully connected layer, and finally an output layer.

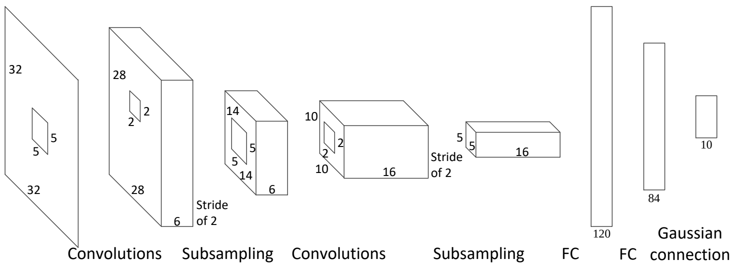

Figure 1. Structure of LeNet-5.

Apart from LeNet-5, we also analyze the latency of AlexNet, ZFNet, and OverFeat. These networks are presented next.

AlexNet [20] is a popular network due to its low top-1 error rate of 37.5% and top-5 error rate of 17% on the ImageNet dataset. Figure 2 shows the structure of AlexNet. There are five convolutional layers in AlexNet, and three max-pooling layers are after the first, second, and last convolutional layers. Finally, there are three fully connected layers.

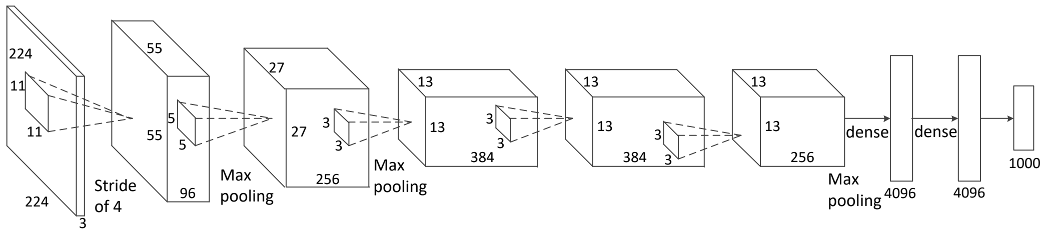

Figure 2. Structure of AlexNet.

ZFNet [21] was the winner of the ImageNet Large Scale Visual Recognition Challenge 2013 (ILSVRC2013). It involved fine-tuning of AlexNet, which reduced the top-5 error rate by 1.7%. The most important contribution of this network is that it proposes a method to visualize the middle layer of convolutional neural networks. Figure 3 shows the structure of ZFNet. The order of the layers in ZFNet is the same as in AlexNet but with an optimized size and stride of some kernels (reduced in the first layer to 7 and 2, respectively).

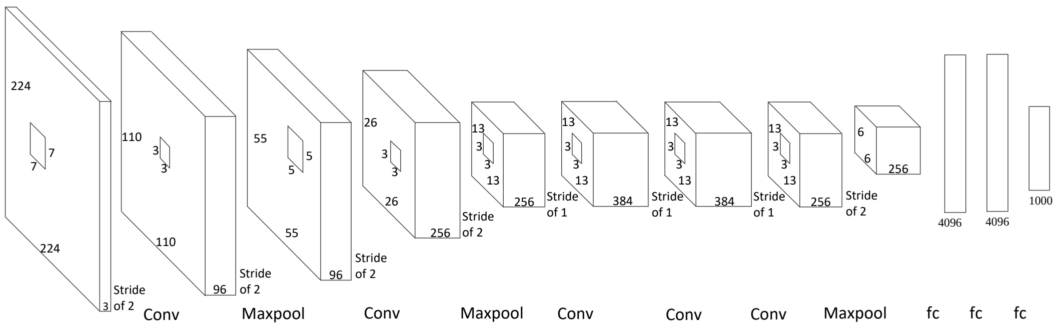

Figure 3. Structure of ZFNet.

OverFeat [22] was the winner of the localization task in ILSVRC2013. It uses the same network for classification, localization, and detection. There are two implementations of OverFeat: a fast model and an accurate model. The fast model has five convolutional layers, three max-pooling layers, and three fully connected layers, while the accurate model has one more convolutional layer. The structures of OverFeat-Fast and OverFeat-Accurate are shown in Figures 4 and 5.

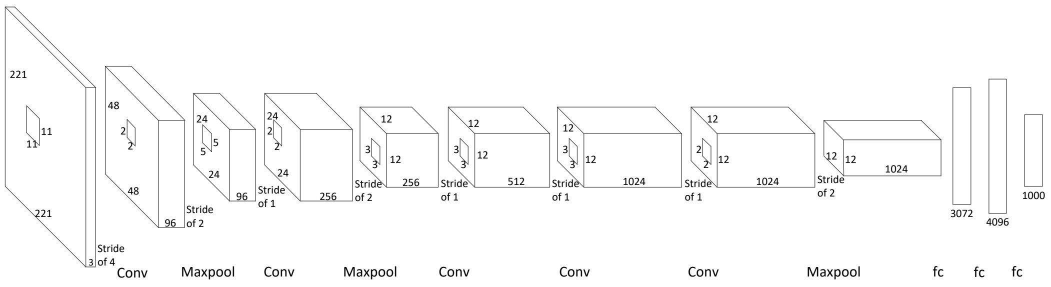

Figure 4. Structure of OverFeat-Fast.

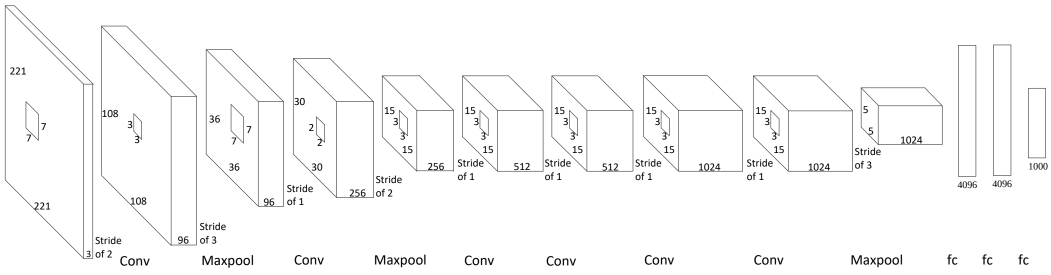

Figure 5. Structure of OverFeat-Accurate.

2.2. HDMI Timing Standard

HDMI (high-definition multimedia interface) is widely used in video transmissions for connecting computers, monitors, and other display devices [23]. A typical timing diagram of HDMI signals is shown in Figure 6. The three main signals used in the HDMI timing standard are Data, HSYNC, and VSYNC. The data signal includes the various pixels of an image, the HSYNC signal indicates the completion of displaying one row of an image and the VSYNC signal indicates the completion of displaying one frame of an image. The HSYNC signal has four main time periods: the horizontal synchronization, the horizontal back porch, the horizontal active video, and the horizontal front porch. The horizontal synchronization represents the start and the end of a given row of pixels in an image. It marks the transition from the end of one row to the beginning of the next. The horizontal active video represents the active video data in one row. The horizontal back porch and the horizontal front porch are the periods of time at the end or the beginning of each horizontal scan line, during which no active video data are transmitted. The VSYNC signal has the same timing periods as HSYNC. In this paper, we use this timing standard for the images and feature maps of our CNN hardware accelerator to provide a clear view of the starting point of one image and the starting point of one new row of the image, which makes the accelerator handle the images more efficiently. Using the HDMI timing standard also provides the hardware accelerator with the possibility of handling video datasets.

2.3. OpenCAPI Interface

OpenCAPI is the open coherent accelerator processor interface, which implements the connection between different components in the computing system, such as the CPU, GPU, and accelerators. OpenCAPI can achieve high-bandwidth, low-latency data transmission between processors and other accelerators. OpenCAPI can ensure coherent data communication between processors and accelerators, reducing the additional overhead required to copy and manage data transfer, enabling a seamless transmission of data. In addition, OpenCAPI provides competitive throughput and latency specs, while reducing the implementation complexity for application developers. Due to these advantages of OpenCAPI, it has been used for high-throughput data-processing applications in the big data analytics domain [24,25]. In this paper, we use OpenCAPI to reduce the latency of data transfer from the host memory to the FPGA memory to improve overall performance.

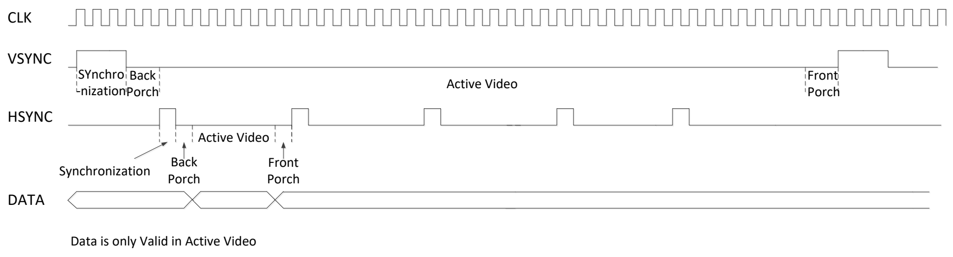

Figure 6. Timing diagram showing the horizontal synchronization and vertical synchronization.

3. Related Work

The acceleration of CNN models is becoming more important due to the need to optimize various model execution parameters, such as latency, throughput, and hardware footprint.

Quantization and model compression methods are widely explored. Lin et al. [26] introduce a fixed-point quantization technique, and they solve the problem of identifying optimal fixed point bit-width allocation across deep convolution network layers. Zhu et al. [27] present optimized binary neural networks, considering the gradient paths, which can lead to a lower error of binary convolutional neural networks. However, these publications do not explore the actual hardware footprint and the challenges specific to mapping quantized networks onto FPGAs. Our paper extends the concept of quantization to FPGA CNN accelerators. We present advanced quantization methods that are optimized for FPGA platforms, enabling efficient deployment of quantized networks while minimizing computational overhead.

Other papers have explored the implementation of convolutional neural networks on hardware. Liu et al. [28] introduce a CNN accelerator for both standard convolution and depthwise separable convolution, which can handle network layers of different scales. Liu et al. [29] propose an accelerator that supports hybrid CNN-SVM algorithms. Ma et al. [30] introduce a CNN accelerator that can handle different kinds of convolutions, especially irregular convolutions, and at the same time, they accelerate the networks by reducing the number of loops in the layers. These accelerators are hardware resource-saving, and they are flexible enough to handle different CNNs. However, their implementation cannot handle different layers coherently, so the latency and throughput are not well optimized. In our design, we pipeline all the layers of the network, which results in a low latency.

Cho et al. [31] propose a CNN accelerator on FPGA; this accelerator utilizes optimized fixed-point data types and employs loop parallelization for optimization. However, their pipeline strategy focuses on increasing throughput by pipelining different images, rather than reducing the latency of processing a single image. In our implementation, we design a pipeline between layers, so the pipeline can benefit both throughput and latency. Li et al. [14] implement all the layers of the whole neural network on one FPGA to increase the throughput. However, they use buffers to store the data between layers, and this will lead to extra pressure on resources. In our design, with the help of the HDMI timing standard we explore in our paper, we do not need to store the results of each layer in a buffer. Instead, we directly send the results to the next layer. In this way, we can use less hardware resources.

OpenCAPI is a framework that can provide a high-bandwidth, low-latency connection between the CPU and the accelerator. Multiple publications make use of OpenCAPI to generate a high-bandwidth accelerator design. Chen et al. [32] use OpenCAPI to accelerate matrix multiplication operations for posit numbers. OpenCAPI is also used to accelerate JSON parsing for big data applications [24]. Peltenburg et al. [33] propose an FPGA accelerator with OpenCAPI to improve the speed at which data can be loaded from disk to memory. Hoozemans et al. [25] explore the benefits of OpenCAPI for FPGA-accelerated big data systems. In this paper, OpenCAPI is used in combination with CNN acceleration to avoid the bottleneck of data transfer.

4. Algorithm Optimization

In this section, we discuss how to optimize the LeNet-5 network to reduce its hardware utilization requirements and allow the hardware design to fit with the target FPGA.

4.1. Improved Model

The convolutional neural network has a large number of parameters, as well as multiplication and addition operations. In our hardware design, all the weights need to be stored on-chip, while all the layers are computed in a pipelined fashion, which requires a lot of FPGA resources. To address this issue, we optimize the LeNet-5 model in terms of reducing input size, quantizing parameters, optimizing activation function, and optimizing output layer calculation method, while ensuring that the model accuracy reduction is minimal.

4.1.1. Reducing Input Size

The input layer of LeNet-5 is 32 × 32, which is larger than the image size in the MNIST dataset. This is done in an attempt to capture obvious features, such as stroke break-point, and to ensure capturing angles in the middle of the highest level of the receptive field. This requires adding padding to the 28 × 28 images before training the network.

In our design, we experiment with reducing the size of the input layer to match that of the dataset images. Experiments with the MNIST dataset demonstrate that these optimizations have a minimal impact on the final prediction accuracy of the network, reducing it by only 0.06%. At the same time, these experiments show that this optimization can reduce the prediction time of the baseline code by up to 16%. In addition, a smaller input size can also save the number of parameters that we need to actively store on the FPGA. When the input size is 32 × 32, the total number of parameters required is 69,564, while when the input size is 28 × 28, the total number of parameters required is 50,004. Therefore, reducing the size of the input features can both reduce parameters and save computation time.

4.1.2. Quantizing Parameters and Data

In this paper, we consider quantizing both the parameters as well as the input data [34–36]. The network uses a single-precision data type with 32 bits to store the parameters, which is quite resource-consuming for hardware, whether for storage or for multiply–add operations. Moreover, such multiply–add operations with high bit width also pose a challenge to the timing of hardware implementation.

To quantize parameters, we follow the method in DoReFa-Net [37], as Equation (1).

$$r_o = 2 * quantize_k\left(\frac{tanh(r_i)}{2 * max(|tanh(r_i)|)} + \frac{1}{2}\right) - 1 \tag{1}$$

where r_i is the full-precision weight. Tanh is used to limit the range of weights to $[-1, 1]$. $quantize_k()$ is a function to quantize its input (which is in the range of [0, 1]) to its nearest neighbor in $\{\frac{i}{2^k-1} | i = 0, \cdots 2^{k-1}\}$ as a k-bit fixed-point in the range of [0, 1]. Finally, r_o is in the range of $[-1, 1]$. In this paper, we quantize the weights to 8-bit fixed points.

For input images, the images in the MNIST dataset are grayscale pictures. The range of input pixels is 0–255, which needs an 8-bit width. However, we can represent this image as a binary (black and white) image, which requires only 1 bit, with a small degradation in the prediction accuracy of only 0.15%. This can save time in transferring the image to the

FPGA. In addition, the first convolutional layer will only need simple logic gates to process the input instead of multipliers, thereby saving a lot of hardware resources.

In this paper, we use the intra-class variance method (Otsu method) [38] to find the optimal threshold to transform the input image into a binary image and then use the binarized image to perform both network training and inference. In our hardware design, we directly input the binarized image into the hardware.

4.1.3. Improved Sigmoid Function

The activation function used by LeNet-5 is the Sigmoid function. The formula is expressed in Equation (2) [39].

$$y(\mathrm{z}) = \frac{1}{1+e^{-z}} \tag{2}$$

The reason the Sigmoid function is used here (rather than other activation functions such as ReLU) is that the output range of the Sigmoid function is limited; it is less affected by data noise, and it is not easy to diverge the data in the process of transmission.

However, using this function directly will cost a lot of hardware resource utilization. According to Equation (2), we can see that the Sigmoid function is a function with an exponential denominator. This kind of function is very difficult to calculate accurately in hardware. Some well-known methods to reduce hardware utilization include look-up tables among others, but these still have high hardware utilization requirements.

The Sigmoid function is essentially a binary classification function, which maps the input to the 0–1 interval. Therefore, this can be approximated as a step function, as shown in Equation (3).

$$\mathrm{f}(x) = \begin{cases} 1 & x > 0 \\ 0 & x \leq 0 \end{cases} \tag{3}$$

The improved Sigmoid function is similar to the function of the activation binarization because they are both step functions. We use a similar method to the training binary activation in [40] to train the improved Sigmoid function. The forward and backward propagations are shown in Equation (4).

$$\begin{aligned} &Forward : F(x) = \begin{cases} 1 & x > 0 \\ 0 & x \leq 0 \end{cases} \\ &Backward : \tfrac{\partial F(x)}{\partial x} = \begin{cases} 2+2x & -1 \leq x < 0 \\ 2-2x & 0 \leq x < 1 \\ 0 & otherwise \end{cases} \end{aligned} \tag{4}$$

The step function is simple to implement in hardware, which reduces the hardware utilization requirements of the network, compared to the original one.

4.1.4. Output Layer Optimization

As discussed earlier, the output layer of LeNet-5 [19] uses RBF to calculate the output classification result. Nowadays, RBF is replaced by a fully connected layer and a SoftMax activation function in the output layers of LeNet [41,42]. The expression of the SoftMax function is shown in Equation (5) [43].

$$y = max_i\left(\frac{e^{x_i}}{\sum_{k=1}^{M} e^{x}k}\right) \tag{5}$$

where x_i is the i_{th} output of the last fully connected layer. M is the total number of the outputs.

Equation (5) calculates the ratio of the exponent of a given x_i to the sum of exponents of all x_i. Then y is calculated as the largest result of Equation (5), which is considered to be the output of the network. This function is used in the training as well as inference of the

CNNs. However, for inference-only purposes, we can simply calculate the largest element as output. Therefore, in this paper, we directly calculate the maximum result from the final fully connected layer and regard its index as the final network output.

4.2. Result of Algorithm Optimization

We use the batch size of 256, the learning rate of 0.001, the epoch number of 40, and an Adam optimizer to train our model. With all the optimization methods shown in Section 4.1, our improved LeNet-5 can achieve an accuracy of 98.8% on the MNIST dataset and an accuracy of 90.2% on the Fashion-MNIST dataset. We also calculate the time of the inference of the LeNet-5 on the MNIST dataset on Pytorch 0.4. The results show that the time to infer one image before the optimization is 6.98 s, and the time to infer one image after all these optimizations is 4.77 s. In total, 31.7% of the operation time has been reduced because of the optimization methods we use.

5. Hardware Design

In this section, we present the overall FPGA hardware structure of FPQNet and the hardware structure of each component. We also introduce several hardware optimization methods.

5.1. Overall Hardware Architecture on OpenCAPI

In this paper, we present a system with ten hardware kernels implementing convolutional neural networks, working together to increase the throughput. We use OpenCAPI to transfer parameters from the host system to the hardware to be shared by all ten kernels. Different images flow into different kernels to be processed in parallel to produce ten different classification results for these images at the same time. Figure 7 shows the architecture of the whole system.

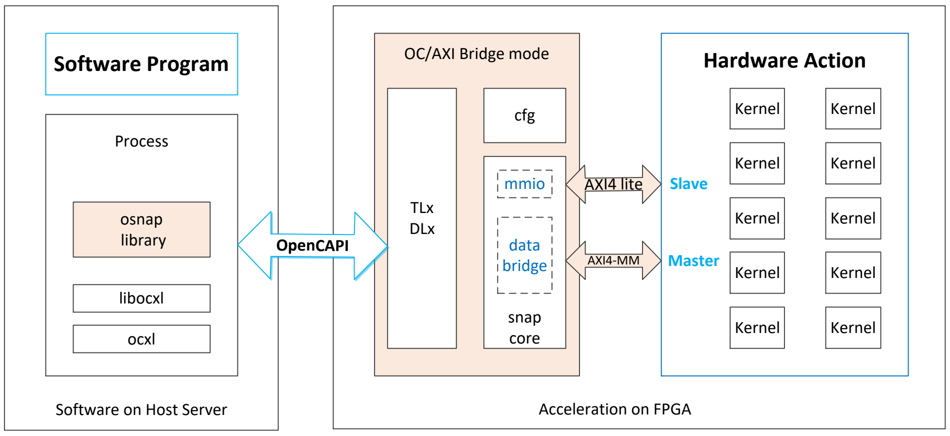

Figure 7. The architecture of the whole system.

5.2. OpenCAPI Data Transfer Module

In this design, the parameters and images are transmitted from the host side of OpenCAPI to the hardware side through the AXI bus. The bit width of the input bus is 512 bits, encoding data in parallel. A Serializer is used to convert the input data to 16 bits. Afterward, there is a data converter to transfer the parameters to the target kernel. These parameters are encoded as 8 bits. The controller in the data converter identifies each of these parameters and transfers them separately to the target kernel. When the images are transferred, the data converter sends the lower 0 to 9 bits of the 16-bit data to each of the 10 kernels, respectively, leaving the remaining 6 bits redundant. After inference, the result from the hardware will be transferred to the host through OpenCAPI.

5.3. Layer-Pipeline Hardware Architecture

In our design, we first transfer the network parameters from the host system through OpenCAPI to the hardware. Afterward, we transfer the images of the dataset to the hardware with OpenCAPI to perform inference measurements on the network. In this project, all the channels in each layer are fully parallelized in each kernel. In addition, we design the first five layers to be pipelined. Since the last two layers only need to compute a limited amount of data, the full network can have ultra-low latency inference capabilities. The process of reading images and performing inference is coherent. The overall timing diagram of the circuit with OpenCAPI is shown in Figure 8.

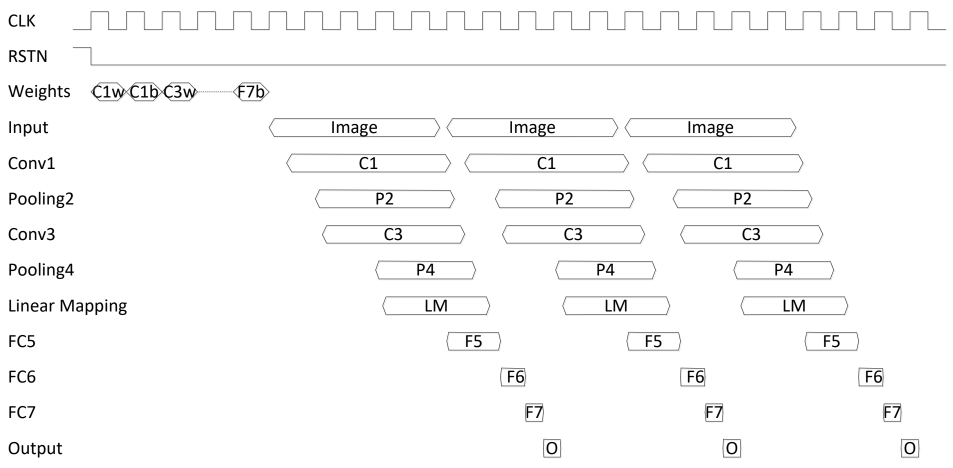

Figure 8. Overall timing diagram of the circuit with OpenCAPI.

With the OpenCAPI interface, we can read the images from the host memory to perform inference on the FPGA directly without having to copy the images to the FPGA memory beforehand. Otherwise, we need to wait for the copy of the images to be completed before inference can start, which will lead to extra latency, as shown in Figure 9. In the design, the latency of copying an image is 5.34 µs, and the latency of the inference operations is 6.17 µs. With OpenCAPI, directly accessing the image from the host memory results in a total latency of 9.32 µs, rather than 11.51 µs. Therefore, without OpenCAPI, the total inference latency would increase by 23.5% as compared to our design.

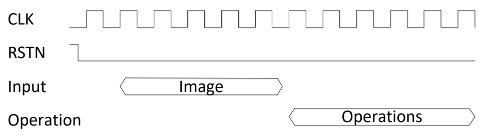

Figure 9. Overall timing diagram of the circuit without OpenCAPI.

The overall schematic diagram of one convolutional neural network is shown in Figure 10. The figure shows the main hardware components of the design, which include the OpenCAPI module used to load parameters and data, convolution modules, pooling modules, linear mapping modules, fully connected modules, and the find max module, to perform predictive classification. We store the parameters on-chip. The intermediate results flow directly into the next layer and do not need to be stored.

5.4. HDMI Timing Standard

In order to ensure proper image transfer, we use the HDMI timing standard, which requires that the pixel points be provided with both line and field synchronization, as shown in Figure 6. This allows the network to process both images as well as video datasets.

The HDMI timing standard is also used in the feature maps of all convolution layers and pooling layers. We generate the VSYNC and the HSYNC signals for each layer to control each operation. When a VSYNC signal comes, we know that a new image starts, so we reset the circuit for a new image. An HSYNC signal indicates a change in the row. When an HSYNC signal arrives, we store the data that follow in a new register to process them using convolution or pooling operations. In this way, each image row is stored in different registers, which makes it easier to start the convolution or pooling operation. The details of the operation of the convolution and the pooling are stated in Sections 5.5 and 5.6. We only use the VSYNC signals and do not use the HSYNC signals in fully connected layers. The VSYNC and HSYNC signals for the output feature maps of each layer can be generated based on the signals of the VSYNC and HSYNC of the input feature maps of the same layer. Our design implements the four main periods in the HDMI standard for each row (horizontal/vertical synchronization, horizontal/vertical back porch, horizontal/vertical active video, and horizontal/vertical front porch), as discussed in Section 2.2. With the VSYNC and HSYNC signals, we know when the next images will be transferred and when the rows finish. The results of the former layer can directly flow into the next layer. In contrast, machine learning synthesis frameworks, such as FINN, have to store intermediate feature maps to extra RAMs to convert the structure of the data. With the direct flow of the feature maps between layers, which is the simplest way of transferring data, unnecessary data migration can be reduced. The HDMI timing standard benefits by both reducing hardware utilization and minimizing data migration.

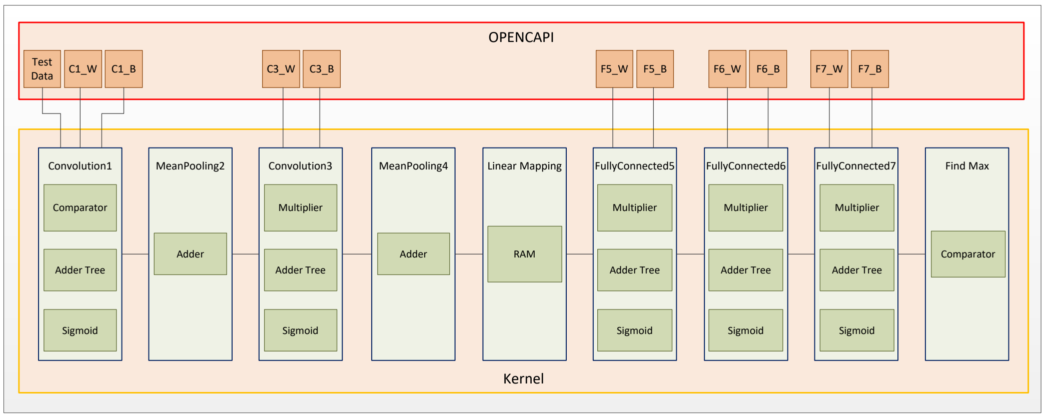

Figure 10. Overall schematic diagram of the hardware design.

5.5. Convolution Layers

For convolutional layer C1, although this layer has convolution operations, it does not need to use multipliers because the input data are binarized. Therefore, it suffices to use simple zero-one comparisons, reassigning the weights to 0 or themselves, as well as simple additions.

The core of this module is a parallel FIFO pipeline designed for 5 × 5 convolution operations. Because real-time input pixels do not need to be stored, only four FIFOs are needed to satisfy the operation as shown in Figure 11. The four FIFOs are connected end to end. The data go into FIFO1 while going into the first group of registers. The data in the registers are used to perform the convolution. Each FIFO holds a whole line of the image, where data output from FIFO1 will flow into FIFO2 and the second group of the registers. The following FIFOs follow the same flow. While the data flow through the FIFOs, the data in the registers are multiplied with the 5 × 5 weights, and the 25 resulting products are summed in an adder tree. The structure of the adder tree is depicted in Figure 12. Since the input data are 0 or 1, selectors instead of multipliers are needed in this layer. There are six modules working in parallel with different output channels in this convolution layer, such that the results of this layer are produced at the same time. When there are only two image output lines, the next pooling layer can start to work.

When it comes to the convolutional layer C3, there are 6 × 16 5 × 5 convolution kernels in this layer. This layer operates by first having the 6 convolution kernels convolve with the 6 sets of output results from the preceding layer, then adding the bias to the results; the

16 sets of convolution kernels operate at the same time. The input of this layer is not only 1, so the multipliers are used in this layer. There are 16 × 6 sets of multipliers that work at the same time.

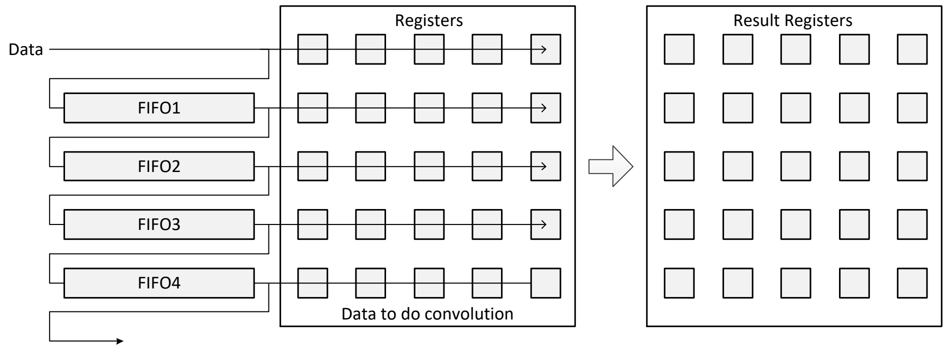

Figure 11. The structure of the FIFOs in the convolution layers.

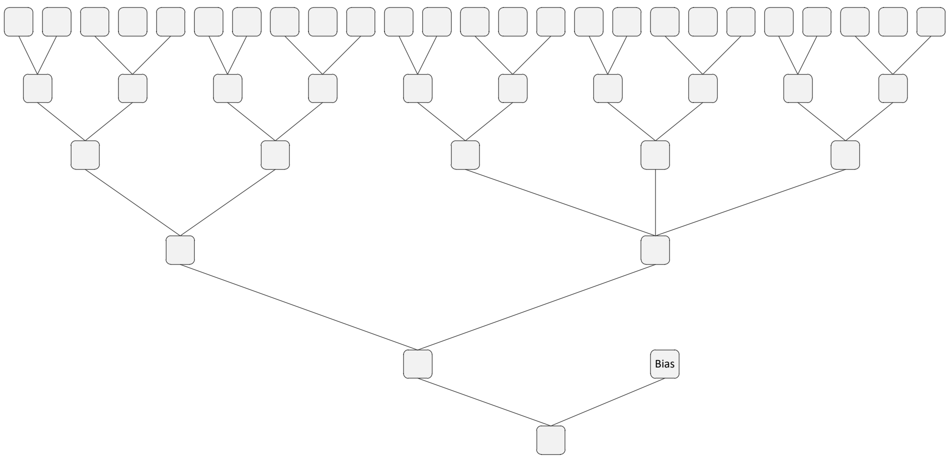

Figure 12. The adder tree in the convolution layers.

5.6. Pooling Layers

This design uses mean pooling with a kernel size of 2. Therefore, FIFOs are also needed in this layer and the principle is the same as the convolution layers, similar to the structure shown in Figure 11. Since the output data of the optimized Sigmoid function activation are only 0 or 1, the average results of this layer have only 5 possible values: 0, 0.25, 0.5, 0.75, and 1. The 5 values of the result are quantified by multiplying by 4, so the outputs of the results are 0, 1, 2, 3, or 4. In this way, we only need to perform addition and do not need to perform division for the average pooling. The pooling layer appears twice in LeNet-5, where it is evaluated in the same way in both cases.

5.7. Fully Connected Layers and Output Layer

When it comes to the C5 layer, this layer is not a convolution operation in the strict sense, so the C5 layer is split into two modules to operate separately. The linear mapping module is the first operation of the C5 layer and the other operation is the fully connected operation.

The main function of linear mapping is to perform mapping on the 16 4 × 4 features from the S4 layer to obtain a 1 × 256 linear feature vector. We first flatten the 4 × 4 data vertically and then connect each group of data first. And finally, we obtain a 1 × 256 feature vector. We use RAM to implement this.

All fully connected layers are the same in this design. Take the C5 layer as an example, the architecture of the fully connected layer is shown in Figure 13. The 1 × 256 vectors from the feature map multiply the 256 × 120 weights and obtain the 1 × 120 results. In this layer, each channel only needs one multiplier and adds each output by the multiplier in real time to obtain the result of the first column. Meanwhile, the 120 multipliers of each channel work at the same time to obtain the final 120 results. The subsequent two fully connected layers are the same as this layer.

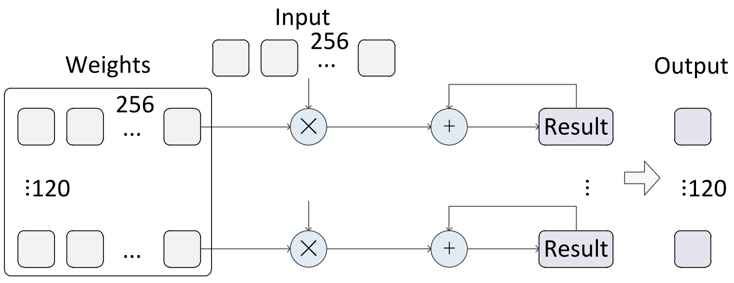

Figure 13. The architecture of the fully connected layer.

As explained in Section 4.1.4, the softmax operation uses a comparator to compare each result and find the maximum result from the outputs of the last fully connected layer. While obtaining the maximum value, the addresses of the data are also marked, which are the final prediction results of the whole design.

6. Experimental Results

In this section, we show our hardware experimental results for our FPQNet implementation. We also compare our results with other solutions and discuss the advantages and disadvantages of our design. Furthermore, we analyze the latency of other popular convolutional neural networks. The code for this work can only be used for this specific LeNet-5 network. However, the same design approach can be used to implement the proposed hardware techniques for other networks.

6.1. Measurement Results

The experimental setup used in this paper to perform the measurements consists of an Inspur FP5290G2 system with a dual-socket POWER9 Lagrange 22-core CPU and OpenCAPI interface to an Aphadata ADM-PCIE-9H7 FPGA board, with a Xilinx XCVU37P chip. Our design is implemented via Verilog hardware design language rather than HLS tools. Our network design runs at a 250 MHz clock frequency. We also use the MNIST dataset for inference measurements.

Measurements show that our design can achieve an ultra-low latency inference time for each MNIST image that is as low as 9.32 µs. Due to the predictable operation of the FPGA designs, all MNIST images run at that exact timing. This indicates that it is possible to use such FPGA ML designs to perform inference in applications that require real-time inference capabilities as well as predictable timing.

The latency to transfer the parameters from CPU to FPGA is 222 µs. However, we only need to transfer the weights once and then can infer multiple images continuously. Therefore, this transfer of parameters is not part of the latency-critical path. When transferring the images and transferring the results back to the CPU for multiple kernels, the high bandwidth of OpenCAPI can avoid a bottleneck.

The hardware utilization values of the FPGA resources for the full designs of one network and ten parallel networks are listed in Table 1. These numbers include the full hardware, including the one or ten convolutional neural network designs, as well as the OpenCAPI interfacing infrastructure. The table shows that the utilization values of the DSPs, BRAMs, and LUTs for one network are 28.97%, 16.91%, and 4.09%. The utilization values of the DSPs, BRAMs, and LUTs for ten networks are 100%, 72.67%, and 81.4%, respectively. One network only uses about one-tenth of the resources on FPGA and multiple networks make almost full use of the resources on FPGA.

As discussed earlier in the paper, the design can achieve an accuracy of 98.8%.

To calculate the throughput, we need to calculate the number of floating-point operations (FLOPs) used by the network first. Now, we calculate the FLOPs of one LeNet-5 convolutional neural network. For convolutional layers, pooling layers, and fully connected layers, we follow the method in [44]. The FLOP numbers for convolutional layers, pooling layers, and fully connected layers are 940,416, 4480, and 83,280, respectively. For the optimized Sigmoid function, each input pixel only needs to undergo one operation, making the total FLOPs for all the Sigmoid functions 4684. For the max operation in the output layer, each input pixel needs to undergo one operation; therefore, the FLOPs of this operation is 10. In total, the number of FLOPs of all operations in one network is 1,032,870, which gives a throughput of 1,032,870/9.32 µs = 110.8 GOPs. Ten neural networks working together make the overall throughput 1108 GOPs. This is the highest throughput reported for accelerating LeNet on FPGAs.

Table 1. Utilization of FPGA resources for the full design.

Resource	Available	Used One Net	Utilization	Used Ten Nets	Utilization
LUTs	1,303,680	53,292	4.09%	1,061,205	81.40%
FFs	2,607,360	94,470	3.62%	1,131,786	43.41%
BRAMs	2016	341	16.91%	1465	72.67%
DSPs	9024	2614	28.97%	9024	100%

From the Vivado report, the power consumption of one LeNet implementation is 420 W. The static power consumption is 37.0 W, which is 9% of the total power consumption and the dynamic power consumption is 383 W, which is 91% of the total power consumption.

6.2. Comparison with Other Solutions

In Table 2, we compare our FPQNet implementation with former implementations of the LeNet network on FPGAs. The accuracy results are all based on inference measurement on the MNIST dataset.

Compared to [45], the latency of 9.32 µs in our implementation is about half of the latency of 18.97 µs in the cited paper. Considering that the frequency of the cited paper is lower than ours, we run a simulation with a reduced frequency equal to that in [45]. This increases our latency to 15.5 µs, which is 18.1% lower than the latency in [45]. Another difference between the two implementations is that the cited paper uses the original dataset image size while we optimize the image by reducing its size. On the other hand, our design is targeted toward video processing; therefore, our design includes the horizontal/vertical back/front porch signals for the images, which is not included in [45]. Due to these differences, it is not possible to provide a direct fair comparison between the latency of the two designs. In addition, our accuracy is 0.4% higher than the accuracy in [45]. However, our implementation uses 1.4 times more LUTs, 12.2 times more DSPs, and 2.8 times more BRAMs. Compared to [46], our latency is about 1000 times lower and our throughput is 264 times higher, despite the fact that they use higher frequencies. At the same time, our accuracy is 1.74% higher than the accuracy in [46]. Reference [47] shows a 53 times higher latency and a 2.5 times lower throughput than our implementation. At the same time, our implementation uses 5.9 times more LUTs and 2.9 times more DSPs but about half of the BRAMs compared to [47].

The table shows that our FPQNet design is able to achieve the lowest latency of all available published solutions, scoring less than half of the lowest latency of any of the alternatives. With 10 networks working together, the overall throughput achieved is almost 25 times higher than other reported designs. However, this comes at the expense of high resource utilization. The resources are used to enable the parallelization needed to achieve lower latency and higher throughput. Still, for modern-day FPGAs, the continued increase in available resources makes such a trade-off viable. In terms of accuracy, with our optimized methods, our design has the highest accuracy of 98.8% among these designs.

Table 2. Comparison with other solutions.

Reference	[45]	[46]	[47]	**This Work**	
				One Net	**Ten Nets**
FPGA platform	Zynq-7020	PYNQ	Virtex7 485t	Alpha Data 9H7	Alpha Data 9H7
Layers	6	7	7	7	7
Frequency (MHz)	150	650	-	250	250
Data precision (bit)	2	-	8	8	8
Latency (µs)	18.97	9100	490	9.32	9.32
Accuracy	98.4%	97.06%	98.16%	98.8%	98.8%
Throughput (GOPs)	-	0.42	44.9	110.8	1108
LUTs	36,798	-	9071	53,292	1061 k
DSPs	214	-	916	2614	9024
BRAMs	123	-	619	341	1465

6.3. Comparison with FINN Solutions

We compare the results in this paper with the results of the same CNN model implemented by FINN [15,48]. We train the same LeNet-5 model with Brevitas. We use the batch size of 256, the learning rate of 0.001, the epoch number of 40, and an Adam optimizer to train the model. The pooling layer is changed from the average pooling layer to the max-pooling layer because we are not able to obtain a good accuracy result using the average pooling layer. After training, we use FINN 0.8.1 to implement the model as well as Alveo U200 FPGA to implement the model. The comparison of the results of the model implemented by FINN and the manually implemented model in this paper is shown in Table 3.

The table shows that—with the same data precision as the LeNet-5 model implemented manually in this paper, which is 8 bits for the weights and 8 bits for the activations—the FINN implementation shows the same accuracy result. The lowest latency that FINN models can achieve is 230.38 µs, which is 24.7× higher compared to the latency of 9.32 µs in our implementation. The LUT utilization of this FINN model is 66,931, which is 25.6% higher than our implementation. The BRAM utilization of the FINN implementation is 32, which is 9.4% of our implementation, and our implementation utilizes 2614 DSPs, whereas the FINN implementation does not employ DSPs.

Table 3. Comparison of our LeNet-5 implementation with FINN.

	FINN	**This Paper**
Model	LeNet-5	LeNet-5
Weight precision (bit)	8	8
Activation precision (bit)	8	8
FPGA platform	Alveo U200	Alpha Data 9H7
Frequency (MHz)	250	250
Accuracy	98.8%	98.8%
Throughput	17,193.95 fps	110.8 GOPs
Latency (µs)	230.38	9.32
BRAM utilization	32	341
LUT utilization	66,931	53,292
DSP utilization	0	2614

We also implement the LeNet-5 models with other bit widths in FINN. The results of the FINN implementations of models with other bit widths of data precision are also shown in Table 4. The lowest latency and the highest throughput that the FINN implementations can achieve are similar. When increasing the bit width of the data precision, the accuracy of the models increases, while the resource utilization of the implementation also increases.

Table 4. FINN results for LeNet-5 models implemented using various other bit widths.

Model	**LeNet-5**	**LeNet-5**	**LeNet-5**	**LeNet-5**
Weight precision (bit)	8	8	2	2
Activation precision (bit)	2	1	2	1
FPGA platform	Alveo U200	Alveo U200	Alveo U200	Alveo U200
Frequency (MHz)	250	250	250	250
Accuracy	98.6%	98.2%	98.6%	97.5%
Throughput (fps)	17,193.95	17,193.95	17,193.95	17,193.95
Latency (µs)	231.02	231.02	231.02	231.02
BRAM utilization	32	32	10	10
LUT utilization	5267	4754	3145	2811

Here, we further explore the reason for the large gap in latency between the results of FINN and the results of our implementation. We look into the latency results of each layer of the CNN model of FINN. The latency of each layer is shown in Table 5. The latency is presented by the number of clock cycles used. As shown in the table, the bottleneck of the design is the first ConvolutionInputGenerator layer, which takes 14,540 clock cycles. The main operation of FINN is based on matrix–vector multiplications. FINN maps each layer of a CNN to a dedicated processing engine named the matrix–vector–threshold unit (MVTU). Every convolutional layer is converted into a sliding window unit (SWU), which generates the image matrix from incoming feature maps, and an MVTU. The ConvolutionInputGenerator in Table 5 is the SWU. The time that FINN needs to generate the input features into the matrix, which is needed for the next step, is long, because it needs to copy data multiple times, and it cannot be parallel when the input only has one channel. However, our design in this paper directly uses FIFOs to store data for the convolution operation, such that data can stream in one by one, and the convolution operation can start and continue to work as long as the kernel size line data are streamed in. In this way, our design does not have the same bottleneck as FINN.

Table 5. The latency of each layer in the FINN-generated model is presented in Table 3.

Layer	**Cycle**
ConvolutionInputGenerator_0	14,540
MatrixVectorActivation_0	3456
StreamingMaxPool_Batch_0	720
ConvolutionInputGenerator_1	9960
MatrixVectorActivation_1	10,240
StreamingMaxPool_Batch_1	80
MatrixVectorActivation_2	7680
MatrixVectorActivation_3	10,080
MatrixVectorActivation_4	840

However, FINN is an automated design framework, which requires less effort to implement a neural network. As an example, for the LeNet-5 designs in this paper, it took us two months to implement the network manually, as compared with two days to implement in FINN.

In conclusion, the results show that FINN can implement a model efficiently and flexibly, and FINN can achieve a trade-off between latency and resource utilization. However, FINN does not focus on implementing extremely low latency implementations. In situations that have strict latency requirements but have sufficient hardware resources and enough human costs, the implementation strategy represented in this paper works better than the FINN implementations.

6.4. Latency Analysis for Other CNNs

The architecture we show in this paper can be used in many other convolutional neural networks, such as AlexNet, ZFNet, and OverFeat. In this section, we estimate the latency we expect to achieve by implementing our architecture to several other networks. Due to the labor-intensive nature of the hardware implementation, we focus on estimating the impact of our architecture rather than on a detailed hardware implementation. As shown in Figure 8, we can calculate how much time is needed in each layer before the next layer can start. The formula of the latency of the convolutional layers is shown in Equation (6), where K is the size of the convolutional kernel, W is the width of the input map, and T is the clock period. Equation (7) shows the latency of the fully connected layers and the output layers, where I is the input length. The total latency is shown in Equation (8), where M is the number of convolutional layers and N is the number of fully connected layers and output layers. With a frequency of 250 MHz and using the ImageNet dataset, the times used in each layer of AlexNet, ZFNet, and OverFeat are shown in Table 6.

$$T_c = KWT \tag{6}$$

$$T_f = IT \tag{7}$$

$$T_{total} = \sum_{i=1}^{M} T_{c\_i} + \sum_{i=1}^{N} T_{f\_i} \tag{8}$$

With our structure, the theoretical latency values of AlexNet, ZFNet, OverFeat-Fast, and OverFeat-Accurate are 69.27 µs, 66.95 µs, 182.98 µs, and 132.6 µs. The longest period of time is spent in fully connected layer 2 and fully connected layer 3. This is because these fully connected layers cannot operate in parallel, and they have to wait for the former layer to finish.

Table 6. Latency analysis for each layer in AlexNet, ZFNet, and OverFeat.

Latency (µs)	**Conv1**	**Pool2**	**Conv3**	**Pool4**	**Conv5**	**Conv6**	**Conv7**
AlexNet	9.86	0.66	0.54	0.32	0.16	0.16	-
ZFNet	6.27	1.32	1.10	0.31	0.16	0.16	-
OverFeat-Fast	10.16	3.84	0.48	0.10	0.14	0.14	-
OverFeat-Accurate	6.19	1.30	1.01	0.36	0.18	0.18	0.18
Latency (µs)	**Conv8**	**Pool9**	**FC1**	**FC2**	**FC3**	**Output**	**Total**
AlexNet	0.16	0.16	0.07	36.86	16.38	4.00	69.27
ZFNet	0.16	0.16	0.07	36.86	16.38	4.00	66.95
OverFeat-Fast	0.14	0.10	0.04	147.46	16.38	4.00	182.98
OverFeat-Accurate	0.18	0.18	0.06	102.40	16.38	4.00	132.6

7. Conclusions

In this paper, we present FPQNet, an ultra-low latency hardware implementation for inference on the LeNet-5 network on FPGAs. In this design, multiple hardware kernels work concurrently to gain high throughput. The design uses several optimization techniques for most network layers, meant to reduce hardware design complexity and improve latency. Furthermore, a pipelining structure is used between most layers to enable the parallel processing of multiple layers. Pipeline optimization with the HDMI timing standard is used to reduce the extra RAM needed between subsequent layers. We also show several quantization methods as well as a simplified hardware-friendly implementation of the Sigmoid function. We demonstrate these methods to implement a hardware design of LeNet-5 for inference on both images as well as video datasets, and compare its latency and accuracy to alternative published solutions. The implementation leverages the OpenCAPI data interface on a POWER9 system to an Alpha Data 9H7 FPGA. For the MNIST dataset, the latency of each image is measured to be as low as 9.32 µs, with an accuracy of 98.8% with a clock frequency of 250 MHz. With ten hardware kernels working concurrently, the overall throughput is 1108 GOPs. The methods proposed in this paper can also be used to improve the performances of other convolutional neural networks. The code for our implementation is open source and publicly available on GitHub at https://github.com/MFJI/FPQNet (accessed on 18 August 2023).

Author Contributions: Methodology, P.H.; Software, M.J.; Validation, Y.C.; Investigation, B.Z.; Writing—original draft, M.J.; Writing—review & editing, Z.A.-A. All authors have read and agreed to the published version of the manuscript.

Funding: This work was supported in part by the Innovation Team Support Plan of Dalian under Grant 2020RT01, and the National Natural Science Foundation of China under Grant 11975066.

Data Availability Statement: The code for our implementation is open source and publicly available on GitHub at https://github.com/MFJI/FPQNet (accessed on 18 August 2023).

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Horng, S.J.; Supardi, J.; Zhou, W.L.; Lin, C.T.; Jiang, B. Recognizing Very Small Face Images Using Convolution Neural Networks. *IEEE Trans. Intell. Transp. Syst.* **2022**, *23*, 2103–2115.
2. Le, D.N.; Parvathy, V.S.; Gupta, D.; Khanna, A.; Rodrigues, J.; Shankar, K. IoT enabled depthwise separable convolution neural network with deep support vector machine for COVID-19 diagnosis and classification. *Int. J. Mach. Learn. Cybern.* **2021**, *12*, 3235–3248.
3. Sharifrazi, D.; Alizadehsani, R.; Roshanzamir, M.; Joloudari, J.H.; Shoeibi, A.; Jafari, M.; Hussain, S.; Sani, Z.A.; Hasanzadeh, F.; Khozeimeh, F.; et al. Fusion of convolution neural network, support vector machine and Sobel filter for accurate detection of COVID-19 patients using X-ray images. *Biomed. Signal Process. Control* **2021**, *68*, 102622.
4. Gao, S.H.; Cheng, M.M.; Zhao, K.; Zhang, X.Y.; Yang, M.H.; Torr, P. Res2Net: A New Multi-Scale Backbone Architecture. *IEEE Trans. Pattern Anal. Mach. Intell.* **2021**, *43*, 652–662.
5. Ye, T.; Zhang, X.; Zhang, Y.; Liu, J. Railway Traffic Object Detection Using Differential Feature Fusion Convolution Neural Network. *IEEE Trans. Intell. Transp. Syst.* **2021**, *22*, 1375–1387.
6. Jung, C.; Han, Q.H.; Zhou, K.L.; Xu, Y.Q. Multispectral Fusion of RGB and NIR Images Using Weighted Least Squares and Convolution Neural Networks. *IEEE Open J. Signal Process.* **2021**, *2*, 559–570.
7. Fukagai, T.; Maeda, K.; Tanabe, S.; Shirahata, K.; Tomita, Y.; Ike, A.; Nakagawa, A. Speed-up of object detection neural network with GPU. In Proceedings of the 25th IEEE International Conference on Image Processing (ICIP), Athens, Greece, 7–10 October 2018; pp. 301–305.
8. Jung, W.; Dao, T.T.; Lee, J. DeepCuts: A Deep Learning Optimization Framework for Versatile GPUWorkloads. In Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI) , Virtual, 20–25 June 2021; pp. 190–205.
9. Ramakrishnan, R.; Dev, K.V.A.; Darshik, A.S.; Chinchwadkar, R.; Purnaprajna, M. Demystifying Compression Techniques in CNNs: CPU, GPU and FPGA cross-platform analysis. In Proceedings of the 34th International Conference on VLSI Design/20th International Conference on Embedded Systems (VLSID), Guwahati, India, 20–24 February 2021; pp. 240–245.
10. Hsieh, M.H.; Liu, Y.T.; Chiueh, T.D. A Multiplier-Less Convolutional Neural Network Inference Accelerator for Intelligent Edge Devices. *IEEE J. Emerg. Sel. Top. Circuits Syst.* **2021**, *11*, 739–750.
11. Zhang, C.; Li, P.; Sun, G.; Guan, Y.; Xiao, B.; Cong, J. Optimizing FPGA-based accelerator design for deep convolutional neural networks. In Proceedings of the 2015 ACM/SIGDA International Symposium On Field-Programmable Gate Arrays, Monterey, CA, USA 22–24 February 2015; pp. 161–170.
12. Liu, L.Q.; Brown, S. Leveraging Fine-grained Structured Sparsity for CNN Inference on Systolic Array Architectures. In Proceedings of the 31st International Conference on Field-Programmable Logic and Applications (FPL), Dresden, Germany, 30 August–3 September 2021; pp. 301–305.
13. Huang, W.J.; Wu, H.T.; Chen, Q.K.; Luo, C.H.; Zeng, S.H.; Li, T.R.; Huang, Y.H. FPGA-Based High-Throughput CNN Hardware Accelerator With High Computing Resource Utilization Ratio. *IEEE Trans. Neural Netw. Learn. Syst.* **2022**, *33*, 4069–4083.
14. Li, H.M.; Fan, X.T.; Jiao, L.; Cao, W.; Zhou, X.G.; Wang, L.L. A High Performance FPGA-based Accelerator for Large-Scale Convolutional Neural Networks. In Proceedings of the 26th International Conference on Field-Programmable Logic and Applications (FPL), Lausanne, Switzerland, 29 August–2 September 2016.

15. Umuroglu, Y.; Fraser, N.J.; Gambardella, G.; Blott, M.; Leong, P.; Jahre, M.; Vissers, K. FINN: A Framework for Fast, Scalable Binarized Neural Network Inference. In Proceedings of the ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, Monterey, CA, USA, 22–24 February 2017; pp. 65–74.
16. Balasubramaniam, S.; Velmurugan, Y.; Jaganathan, D.; Dhanasekaran, S. A Modified LeNet CNN for Breast Cancer Diagnosis in Ultrasound Images. *Diagnostics* **2023**, *13*, 2746.
17. Yuan, Y.X.; Peng, L.N. Wireless Device Identification Based on Improved Convolutional Neural Network Model. In Proceedings of the 18th IEEE International Conference on Communication Technology (IEEE ICCT), Chongqing, China, 8–11 October 2018; pp. 683–687.
18. Dubey, A. Agricultural plant disease detection and identification. *Int. J. Electr. Eng. Technol.* **2020**, *11*, pp. 354–363.
19. Lecun, Y.; Bottou, L.; Bengio, Y.; Haffner, P. Gradient-based learning applied to document recognition. *Proc. IEEE* **1998**, *86*, 2278–2324.
20. Krizhevsky, A.; Sutskever, I.; Hinton, G.E. ImageNet Classification with Deep Convolutional Neural Networks. *Commun. ACM* **2017**, *60*, 84–90.
21. Zeiler, M.D.; Fergus, R. Visualizing and Understanding Convolutional Networks. In Proceedings of the 13th European Conference on Computer Vision (ECCV), Zurich, Switzerland, 6–12 September 2014; pp. 818–833.
22. Sermanet, P.; Eigen, D.; Zhang, X.; Mathieu, M.; Fergus, R.; LeCun, Y. Overfeat: Integrated recognition, localization and detection using convolutional networks. *arXiv* **2013**, arXiv:1312.6229.
23. Song, T.L.; Jeong, Y.R.; Yook, J.G. Modeling of Leaked Digital Video Signal and Information Recovery Rate as a Function of SNR. *IEEE Trans. Electromagn. Compat.* **2015**, *57*, 164–172.
24. Peltenburg, J.; Hadnagy, A.; Brobbel, M.; Morrow, R.; Al-Ars, A. Tens of gigabytes per second JSON-to-Arrow conversion with FPGA accelerators. In Proceedings of the 20th International Conference on Field-Programmable Technology (ICFPT), Auckland, New Zealand, 6–10 December 2021; pp. 194–202.
25. Hoozemans, J.; Peltenburg, J.; Nonnemacher, F.; Hadnagy, A.; Al-Ars, Z.; Hofstee, H.P. FPGA Acceleration for Big Data Analytics: Challenges and Opportunities. *IEEE Circuits Syst. Mag.* **2021**, *21*, 30–47.
26. Lin, D.D.; Talathi, S.S.; Annapureddy, V.S. Fixed Point Quantization of Deep Convolutional Networks. In Proceedings of the 33rd International Conference on Machine Learning, New York, NY, USA, 19–24 June 2016.
27. Zhu, B.; Hofstee, P.; Lee, J.; Alars, Z. Improving Gradient Paths for Binary Convolutional Neural Networks, BMVC 2022. Available online: https://bmvc2022.mpi-inf.mpg.de/0281.pdf (accessed on 18 August 2023).
28. Liu, B.; Zou, D.Y.; Feng, L.; Feng, S.; Fu, P.; Li, J.B. An FPGA-Based CNN Accelerator Integrating Depthwise Separable Convolution. *Electronics* **2019**, *8*, 281.
29. Liu, B.; Zhou, Y.Z.; Feng, L.; Fu, H.S.; Fu, P. Hybrid CNN-SVM Inference Accelerator on FPGA Using HLS. *Electronics* **2022**, *11*, 2208.
30. Ma, Y.F.; Cao, Y.; Vrudhula, S.; Seo, J.S. Optimizing the Convolution Operation to Accelerate Deep Neural Networks on FPGA. *IEEE Trans. Very Large Scale Integr. (Vlsi) Syst.* **2018**, *26*, 1354–1367.
31. Cho, M.; Kim, Y. FPGA-Based Convolutional Neural Network Accelerator with Resource-Optimized Approximate Multiply-Accumulate Unit. *Electronics* **2021**, *10*, 2859.
32. Chen, J.Y.; Al-Ars, Z.; Hofstee, H.P. A Matrix-Multiply Unit for Posits in Reconfigurable Logic Leveraging (Open) CAPI. In Proceedings of the Conference on Next Generation Arithmetic (CoNGA), Singapore, 28 March 2018.
33. Peltenburg, J.; van Leeuwen, L.T.J.; Hoozemans, J.; Fang, J.; Al-Ars, A.; Hofstee, H.P.; Soc, I.C. Battling the CPU Bottleneck in Apache Parquet to Arrow Conversion Using FPGA. In Proceedings of the 19th International Conference on Field-Programmable Technology (ICFPT), Maui, HI, USA, 9–11 December 2020; pp. 281–286.
34. Zhu, B.Z.; Al-Ars, Z.; Pan, W. Towards Lossless Binary Convolutional Neural Networks Using Piecewise Approximation. In Proceedings of the 24th European Conference on Artificial Intelligence (ECAI), European Assoc Artificial Intelligence, Santiago de Compostela, Spain, 29 August–8 September 2020; pp. 1730–1737.
35. Zhu, B.Z.; Al-Ars, Z.; Hofstee, H.P. NASB: Neural Architecture Search for Binary Convolutional Neural Networks. In Proceedings of the International Joint Conference on Neural Networks (IJCNN) held as part of the IEEE World Congress on Computational Intelligence (IEEE WCCI), Glasgow, UK, 19–24 July 2020.
36. Baozhou, Z.; Hofstee, P.; Lee, J.; Al-Ars, Z. SoFAr: Shortcut-based fractal architectures for binary convolutional neural networks. *arXiv* **2020**. arXiv:2009.05317.
37. Zhou, S.; Wu, Y.; Ni, Z.; Zhou, X.; Wen, H.; Zou, Y. Dorefa-net: Training low bitwidth convolutional neural networks with low bitwidth gradients. *arXiv* **2016**, arXiv:1606.06160.
38. Otsu, N. Threshold Selection Method From Gray-Level Histograms. *IEEE Trans. Syst. Man Cybern.* **1979**, *9*, 62–66.
39. Han, J.; Moraga, C. The influence of the sigmoid function parameters on the speed of backpropagation learning. In *International Workshop on Artificial Neural Networks*; Springer: Berlin/Heidelberg, Germany, 1995; pp. 195–201.
40. Liu, Z.C.; Luo, W.H.; Wu, B.Y.; Yang, X.; Liu, W.; Cheng, K.T. Bi-Real Net: Binarizing Deep Network Towards Real-Network Performance. *Int. J. Comput. Vis.* **2020**, *128*, 202–219.
41. Givaki, K.; Salami, B.; Hojabr, R.; Tayaranian, S.M.R.; Khonsari, A.; Rahmati, D.; Gorgin, S.; Cristal, A.; Unsal, O.S.; Soc, I.C. On the Resilience of Deep Learning for Reduced-voltage FPGAs. In Proceedings of the 28th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP), Vasteras, Sweden, 11–13 March 2020; pp. 110–117.

42. Wang, H.; Wang, Y.T.; Zhou, Z.; Ji, X.; Gong, D.H.; Zhou, J.C.; Li, Z.F.; Liu, W. CosFace: Large Margin Cosine Loss for Deep Face Recognition. In Proceedings of the 31st IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR), Salt Lake City, UT, USA, 18–23 June 2018; pp. 5265–5274.
43. Qiao, S.J.; Ma, J. FPGA Implementation of Face Recognition System Based on Convolution Neural Network. In Proceedings of the Chinese Automation Congress (CAC), Xian, China, 30 November–2 December 2018; pp. 2430–2434.
44. Molchanov, P.; Tyree, S.; Karras, T.; Aila, T.; Kautz, J. Pruning convolutional neural networks for resource efficient inference. *arXiv* **2016**, arXiv:1611.06440.
45. Zhang, L.; Bu, X.; Li, B. XNORCONV: CNNs accelerator implemented on FPGA using a hybrid CNNs structure and an inter-layer pipeline method. *IET Image Process.* **2020**, *14*, 105–113.
46. Laguduva, V.R.; Mahmud, S.; Aakur, S.N.; Karam, R.; Katkoori, S. Dissecting convolutional neural networks for efficient implementation on constrained platforms. In Proceedings of the 2020 33rd International Conference on VLSI Design and 2020 19th International Conference on Embedded Systems (VLSID), Bangalore, India, 4–8 January 2020; pp. 149–154.
47. Li, Z.; Wang, L.; Guo, S.; Deng, Y.; Dou, Q.; Zhou, H.; Lu, W. Laius: An 8-bit fixed-point CNN hardware inference engine. In Proceedings of the 2017 IEEE International Symposium on Parallel and Distributed Processing with Applications and 2017 IEEE International Conference on Ubiquitous Computing and Communications (ISPA/IUCC), Guangzhou, China, 12–15 December 2017; pp. 143–150.
48. Blott, M.; Preusser, T.B.; Fraser, N.J.; Gambardella, G.; K. O'Brien, Umuroglu, Y.; Leeser, M.; Vissers, K. FINN-R: An End-to-End Deep-Learning Framework for Fast Exploration of Quantized Neural Networks. *ACM Trans. Reconfig. Technol. Syst.* **2018**, *11*, 1–23.