electronics

MDPI

Article

Quantization-Based Optimization Algorithm for Hardware Implementation of Convolution Neural Networks

Bassam J. Mohd [1,*,†,‡], Khalil M. Ahmad Yousef [1,‡], Anas AlMajali [1,‡] and Thaier Hayajneh [2,‡]

[1] Department of Computer Engineering, Faculty of Engineering, The Hashemite University, Zarqa 13133, Jordan; khalil@hu.edu.jo (K.M.A.Y.); almajali@hu.edu.jo (A.A.)
[2] Department of Computer and Information Sciences, Fordham University, New York, NY 10023, USA; thayajneh@fordham.edu
* Correspondence: bassam@hu.edu.jo; Tel.: +962-5-390-3333
† Current address: Department of Computer Engineering, Faculty of Engineering, The Hashemite University, Zarqa 13133, Jordan.
‡ These authors contributed equally to this work.

Abstract: Convolutional neural networks (CNNs) have demonstrated remarkable performance in many areas but require significant computation and storage resources. Quantization is an effective method to reduce CNN complexity and implementation. The main research objective is to develop a scalable quantization algorithm for CNN hardware design and model the performance metrics for the purpose of CNN implementation in resource-constrained devices (RCDs) and optimizing layers in deep neural networks (DNNs). The algorithm novelty is based on blending two quantization techniques to perform full model quantization with optimum accuracy, and without additional neurons. The algorithm is applied to a selected CNN model and implemented on an FPGA. Implementing CNN using broad data is not possible due to capacity issues. With the proposed quantization algorithm, we succeeded in implementing the model on the FPGA using 16-, 12-, and 8-bit quantization. Compared to the 16-bit design, the 8-bit design offers a 44% decrease in resource utilization, and achieves power and energy reductions of 41% and 42%, respectively. Models show that trading off one quantization bit yields savings of approximately 5.4K LUTs, 4% logic utilization, 46.9 mW power, and 147 µJ energy. The models were also used to estimate performance metrics for a sample DNN design.

Keywords: convolutional neural networks; deep neural networks; FPGA; hardware design; quantization; resource-constrained devices

Citation: Mohd, B.J.; Ahmad Yousef, K.M.; AlMajali, A.; Hayajneh, T. Quantization-Based Optimization Algorithm for Hardware Implementation of Convolution Neural Networks. *Electronics* **2024**, *13*, 1727. https://doi.org/10.3390/electronics13091727

Academic Editor: Chen Yang

Received: 5 March 2024
Revised: 20 April 2024
Accepted: 26 April 2024
Published: 30 April 2024

1. Introduction

Convolutional neural networks (CNNs) have been successfully applied in a wide range of cognitive tasks such as image recognition and classification, natural language processing, object detection, voice recognition, and autonomous driving [1,2]. This is mainly due to CNN's remarkable accuracy and performance. However, CNNs require significant computations and memory resources. Recent improvements in CNN accuracy have been achieved by over-parameterized models, which present a significant challenge when being deployed in resource-constrained applications [1,3].

The hardware implementation of CNNs has been studied and optimized in recent years. One motivation to implement CNN in hardware is to achieve the desired performance (i.e., in terms of high-accuracy and real-time response) with the least power and energy consumption. In fact, it is currently trendy to see CNN designs being implemented in digital devices, embedded systems, and edge devices; which are resource constrained [4,5]. However, with such CNN implementations on resource-constrained devices (RCDs) and as the number of neurons grows, the area, energy, and delay grow

quadratically, and memory traffic increases significantly. Thus, realizing large CNNs in hardware is a significant challenge [6].

RCDs, which are typically found in embedded systems and edge devices, have limited resources that include small memory (RAM and ROM) to execute the applications and store data/parameters, low data processing power, limited battery capacity, and small physical area [7]. Consequently, designing CNNs for RCDs is challenging and requires a delicate balance between resources and model accuracy, and even unaffordable for larger CNN models [4,8,9].

While it is important to streamline the CNN design, CNN layers have different design challenges and present a unique hurdle to the optimum method(s) that reduces CNNs complexity. On one hand, the convolutional layers are computation-centric as they contain a high volume of computations; and on the other hand, the fully connected (FC) layers are memory-centric and require large memory bandwidth. An efficient optimization method should focus on simplifying the computations and the memory system [5].

Several optimization methods have been proposed to reduce the design complexity and simplify the implementation of CNNs. One such method is quantization, which involves replacing real-valued numbers with low-bit (i.e., low-precision) fixed-point integers. By reducing the bit width of parameters (i.e., weights and biases) and the outputs from each layer (activations), quantization unlocks speed gains, lower power consumption, and smaller memory footprints.

Numerous research has reported quantized CNNs using 32-, 16- and 8-bits [5,6,10–15]. Additionally, other research work proposed aggressive quantization techniques that constrain the parameters to as low as two bits (e.g., ternary networks) or even one bit (e.g., binarized networks). Aggressive quantization, however, potentially degrades CNN accuracy [16]. To compensate for the accuracy drop, the number of neurons is increased, which, unfortunately, could offset quantization benefits [17].

Another proposed method (to reduce CNN design complexity) involves approximating low-rank filters in pre-trained networks. For example, Zhang et al. [18] proposed an approach that enables an asymmetric reconstruction, which reduces the rapidly accumulated error when multiple layers are approximated. Other methods restrict filters with low-rank constraints during the training phase. For example, the low-rank tensor decomposition proposed in [19] removes the redundancy in the weights of a pre-trained neural network (NN) and reduces the number of parameters significantly.

Another proposed optimization method involves employing smaller and more efficient CNN architectures. One idea involves replacing 3×3 convolutional filters with 1×1 size to reduce the model complexity, as demonstrated in GoogLeNet [20] and SqueezeNet [21]. A second idea involves utilizing residual connections to relieve the gradient vanishing problem during deep network training, such as the design in ResNet [22]. A third idea involves generalizing the group convolution and the depthwise separable convolution, such as the design in ShuffleNet [23]. In fact, the utilized depthwise separable convolutions, as in Xception [24] and MobileNet [25], have been proved to be effective.

Other methods to reduce CNN complexity involve pruning and structural sparsity. Pruning reduces model size by removing redundant weights [26]. It is typically achieved by setting some weights, neurons, or connections in the NN model to zero or near-zero values [27]. Han et al. [28] discussed pruning to reduce the memory requirement of CNNs with no loss of accuracy. Additionally, the research works in [29,30] employ structural sparsity for more energy-efficient compression. Further, singular value decomposition (SVD) is used, as in [5], to reduce memory footprint.

Several technologies are used to realize CNN implementation, including CPUs, GPUs, FPGAs, or ASIC. However, FPGAs are believed to present attractive implementations compared to other technologies for several reasons [1,2,31]:

- FPGAs are more energy efficient compared with GPUs and CPUs.
- FPGAs have parallel computing resources with high performance.

- Reconfigurability in FPGAs provides significant flexibility to explore CNNs' design options and alternatives.
- FPGAs provide high security [31].

While deep neural networks (DNNs) continue to be a major research focus, there is a growing enthusiasm for smaller and lightweight NNs. Future FPGA-based NN acceleration will be embedded, lightweight, and portable [1]. Additionally, the lightweight CNN quantization method extends further to optimize DNNs. By strategically applying it to specific layers, we can minimize quantization errors and efficiently implement the DNN using fewer FPGAs, leading to significant power and energy savings. Hence, it is important to focus FPGA-based research on lightweight CNNs, which are targeted for embedded designs and applications such as AI-of-Things (AIoT) [1] and are also beneficial for DNNs.

The main objective of this research is to develop an FPGA-based quantization method to reduce CNN hardware design, and to estimate-and-model the enhancements in area, power, and energy. This will enable implementing CNNs in RCDs (e.g., FPGA devices). Additionally, the derived models will facilitate the quick and easy evaluation of DNN design choices. Our research successfully addressed the above-stated objective by delivering the following key contributions.

- Designed and validated an algorithm to quantize the full CNN model. The novelty of the algorithm is that it combines quantization-aware training (QAT) and post-training quantization techniques (PQT), and it provides full model quantization (weights and activations) without increasing the number of neurons.
- Explored, designed, and verified multiple hardware designs of the quantized CNN; each design with different quantization bits to fit the FPGA capacity and resources.
- Performed compilation and synthesis of the hardware designs in the FPGA device, which is Altera Stratix IV.
- Analyzed and modeled the resources, timing, throughput, power, and energy results.
- Estimated the performance metrics for a sample DNN design using the derived models.

The rest of the paper is organized as follows. Section 2 summarizes the latest research work in CNN quantization. Section 3 presents the research methodology and design flow. Section 4 explains the proposed algorithm in detail. Section 5 discusses the FPGA implementation of the quantized CNN model. Section 6 summarizes the area, timing, throughput, power, and energy results of FPGA implementations. Section 7 discusses modeling metrics and applying them to DNN. Section 8 discusses concluding remarks and future research opportunities.

2. Related Work

In general, examining research work in CNN quantization, researchers have exercised two design options when quantizing a CNN model:

- Quantization of parameters only versus quantization of the entire model (i.e., parameters and activations) [26];
- QAT versus PTQ. Generally speaking, with the same bit precision, QAT achieves higher accuracy [32].

It is noteworthy to mention that the approach of "quantization of parameters only" somewhat strikes a balance between "no quantization", which has expensive computations and high memory bandwidth, and "quantization of the entire model" that unleashes the full potential of quantization, but tends to be harder and requires careful implementation due to potential higher accuracy loss.

Additionally, we should mention that QAT involves a cyclical process of quantization and retraining to optimize a pre-trained CNN for quantized representations. The key steps, which are repeated until convergence, are as follows: quantizing the weights, forward pass using floating point activations, and lastly, back-propagation pass using floating point gradient. Because of the weight quantization, it is necessary to approximate the gradients using (for example) straight-through estimator (STE) [33].

From the implementation point of view, the reader should notice that high accuracy QAT requires a long training time; however, it is justifiable for the models that will be deployed for long periods of time. On the other hand, PTQ quantizes a pre-trained NN without any extra training, which is helpful when training data are not sufficient or unavailable, and quickly provides quantized NN. In both QAT and PTQ, sometimes quantization can be fine-tuned with a small set of calibration data [32].

Several quantization-based frameworks and tools were developed to automate and optimize NN implementation on FPGAs and ASICs [34]. Examples include the Xilinx FINN [35] and hls4ml [36] frameworks. On one hand, the FINN framework leverages Brevitas [37] for quantization-aware training. A research example, which explored the trade-off between accuracy and resource usage on FPGAs using such a framework is Ducasse et al. [34]. This research implemented quantized NNs (with less than 8 bits) in FPGAs while maintaining 88% accuracy. On the other hand, hls4ml is an open-source framework that combines software and hardware design to translate NNs to FPGAs and ASICs. Its workflow integrates network optimization techniques, allowing for low-power implementations. While such frameworks are valuable for exploring efficient hardware solutions, it is important to consider their limitations such as customization and troubleshooting.

Researchers also use algorithm hardware co-design technique to automatically develop efficient NNs and hardware, which considers both software and hardware together during the design and quantization process. For example, Fan et al. [38] proposed a three-phase co-design framework, which decouples training from the design space exploration and adopts the Gaussian process to predict accuracy and power consumption. In comparison with the manually-designed ResNet101, InceptionV2, and MobileNetV2, the authors' proposed framework can achieve up to 5% higher accuracy with up to 3× speed up on the ImageNet dataset. Additionally, Wang et al. [39] proposed a hardware/software co-design methodology targeting CPU+FPGA-based heterogeneous platforms. The authors' proposed methodology includes optimization (i.e., hardware-aware NN pruning, clustering and quantization), and an end-to-end design space exploration flow. Experimental results show that the authors' work can achieve a peak throughput of 2.13 TOPS. Moreover, Haris et al. [40] proposed SECDA, a hardware/software co-design methodology, which combines the system's simulation with FPGA execution. SECDA achieved an average performance speedup across models of up to 3.5× with a 2.9× reduction in energy consumption over CPU-only inference. Overall, we believe that a major challenge in algorithm-hardware co-design is the high cost of training NNs and the lengthy process of implementing hardware. This makes it impractical to explore the vast range of potential NN architectures and hardware designs [27,38].

The rest of this section discusses research in "parameter quantization only" or simply to be called "parameter quantization", and "entire model quantization".

2.1. Parameter Quantization

Compared with quantization of the entire NN model, parameter quantization is relatively easier in implementation and demonstrates high accuracy results [9,41,42]. For example, Holt et al. [43] performed training with fixed point 16-bit weights and 8-bit inputs. While parameter quantization reduces model size with minimal accuracy loss (i.e., by a few percent drop) for 32-bit and 16-bit precision, some researchers explore even lower bit widths, accepting a larger accuracy drop for further efficiency gains. For example, the binarized NN (BNN) approach constrains the weights to one bit, which can represent two possible values (e.g., -1 or 1). This approach simplifies the hardware by replacing multiply-accumulate units with accumulate units [44]. There are Ternary networks [9], which increase weight bit width to 2 bits representing three values (e.g., $-1, 0, 1$). The additional bit improves accuracy compared with BNN [9].

There is also a three-bit quantization of parameters examined as well. For example, Park et al. [45] developed an FPGA-based fixed-point NN using only on-chip memory,

by quantizing weights to 3 bits (for input and hidden layers) and 8 bits (for output layer). The authors utilized QAT, where the training is performed in three steps: floating-point training, optimal uniform quantization, and retraining with fixed-point weights. The FPGA implementation is tested for the Modified National Institute of Standards and Technology (MNIST) handwritten digit recognition benchmark and a phoneme recognition task on Texas Instruments/Massachusetts Institute of Technology (TIMIT) corpus. The implemented FPGA shows a throughput that is about one-quarter of a GPU system, but consumes 2~4% of the energy consumption of the GPU system, resulting in over 10 times the power efficiency. One issue of the design is that the base design (with floating point prior to quantization) has an error rate up to 27.81%, which is significant and potentially affects the quantized design results.

Zhou et al. [42] proposed an incremental network quantization (INQ) method to quantize model weights to either zero value or power of two values. The method consists of three operations: weight partition, group-wise quantization, and re-training. Weights in each layer are divided into two groups. The first group is from a low-precision base and quantized by a variable-length encoding method. The second group is retrained to compensate for the accuracy loss from the quantization. The operations are repeated until all the weights are converted into low-precision ones.

Abdelouahab et al. [17] presented a method to optimize DSP utilization in FPGAs for CNN implementations while maintaining high accuracy. The method is based on exploring design space by varying the neuron count in each layer and the precision of weights and biases. The study concluded that classification accuracy increases with the number of neurons and with numerical precision.

2.2. Quantize the Entire Model: Weights and Activations

Despite the complexities of full model quantization, numerous works implemented, examined, and implemented this method. For example, the 16-bit quantization Q6.10 (6-bit integer and 10-bit fraction) is used as in [6,46], and quantization Q8.8 (8-bit integer and 8-bit fraction) is used as in [47]. Additionally, as in [48], 32-bit quantization is used during training while 16-bit is used in the inference phase.

Courbariaux et al. [49] trained a set of CNNs on three benchmark datasets, with three formats: floating point, fixed point, and dynamic fixed point. The research concluded that the minimum bit-width for the activations is 10-bit and 12-bit for parameters. Below these bit-widths, the error rate rises significantly.

Vanhoucke et al. [50] performed the training using a single precision floating point, while evaluation was performed with 8-bit weights and activations. When compared with floating-point, the 8-bit arithmetic provided an over two times speedup without having a loss of accuracy when being applied on speech recognition design.

Qiu et al. [5] presented an FPGA design for large-scale CNN. The proposed method combines SVD and quantization. The authors' implementation of the method consists of two phases: weight quantization and data quantization. The method reduces fraction bits in weights and activations while minimizing accuracy loss. Compared with 16-bit quantization, 8/4-bit quantizations halve the storage space for intermediate data and reduce three-fourths of the memory footprint. However, when 8/4-bit quantizations are used, a 0.4% accuracy loss is introduced by data quantization for the VGG16 model. One issue of this work is that the accuracy of the base design (i.e., VGG16 with floating point) is 68.1%. As such, the reported accuracy loss might not reflect the loss in the quantization accuracy.

Chen et al. [6] presented a detailed study in hardware accelerators for small and large NNs. For small NNs, all neurons and synapses are realized in hardware. Small designs and short interconnects result in high-speed and low-energy implementation. The study shows that, as the number of neurons grow, the area, energy, and delay grow quadratically, which makes realizing large NNs in hardware very challenging. Another issue for large-scale NNs is the high memory traffic. Hence, the hardware implementation for large-scale NNs realizes a fraction of neurons and synapses. For hardware implementation,

the research suggested the following techniques: tiling, pipelining, buffers, and DMAs. It also replaced the floating-point computations with 16 fixed-point computations (6 bits for the integer part, 10 bits for the fractional part), which reduces the power and area and slightly increases the error rate by 0.26%. The CNN was trained and tested on the standard MNIST machine-learning benchmark [51], and the CNN implementation leverages a 65 nm process technology.

2.3. Discussion

In light of the above research, we believe it is important to highlight the following points.

- Parameter quantization reduces CNN complexity efficiently, but misses out on potential reductions from activation quantization. Full model quantization targets both computation and memory but requires careful accuracy consideration.
- Another issue is the accuracy of the base design (i.e., with floating point and prior to quantization). In some research works, the accuracy of the base design should have been better, which indicates issues with model training [5,45].
- Some of the quantization methods add additional neuron operations to mitigate the accuracy drop, which in turn increases model complexity [17].
- Large NN implementation techniques (e.g., tiling and pipelining) are different from small NNs [6].
- Surprisingly, most research on quantized designs fail to model key performance metrics like power and energy.

Our focus in this research is to present a comprehensive approach to the quantization method. It includes:

- Proposing a full-model quantization algorithm without requiring extra neurons;
- Modeling the impact of quantization on resources (e.g., energy, power, and area);
- It should scale up to DNNs.

To the best of our knowledge, our literature review found no prior work that encompassed such a comprehensive approach.

3. Research Methodology

In this section, we discuss our research methodology, which has a similar flow to those used in other similar research works, including [52–54]. Figure 1 outlines the flow steps, which mainly consist of algorithm development and simulations, hardware design, and performance evaluation. The following subsections provide a detailed explanation of each step in Figure 1.

3.1. Research Objectives

Clear research objectives should be set early in the project because they direct the activities and tasks of the research. One objective of this research is to develop a method to reduce the CNN hardware design. Quantization was chosen because it is effective and powerful. Another objective is to devise an algorithm that can be applied to CNNs. We are aware of numerous research works reporting the quantization of a wide range of bits; however, it is important to present an algorithm that could be applied to any CNN. Another objective is to understand the impact of quantization on resources, timing, throughput, power, and energy, which is achieved through modeling those metrics. Such understanding would enable implementing CNNs on RCDs (e.g., FPGA devices), as well as on DNNs.

Finally, determining the optimal bit-width for a specific CNN while maintaining accuracy is a common objective, and extensive research has been conducted on this aim. Instead, we challenge this approach by offering a more versatile method (i.e., a quantization algorithm and modeling of performance metrics) that can be seamlessly applied to any NN and DNN. This empowers broader applicability and adaptability.

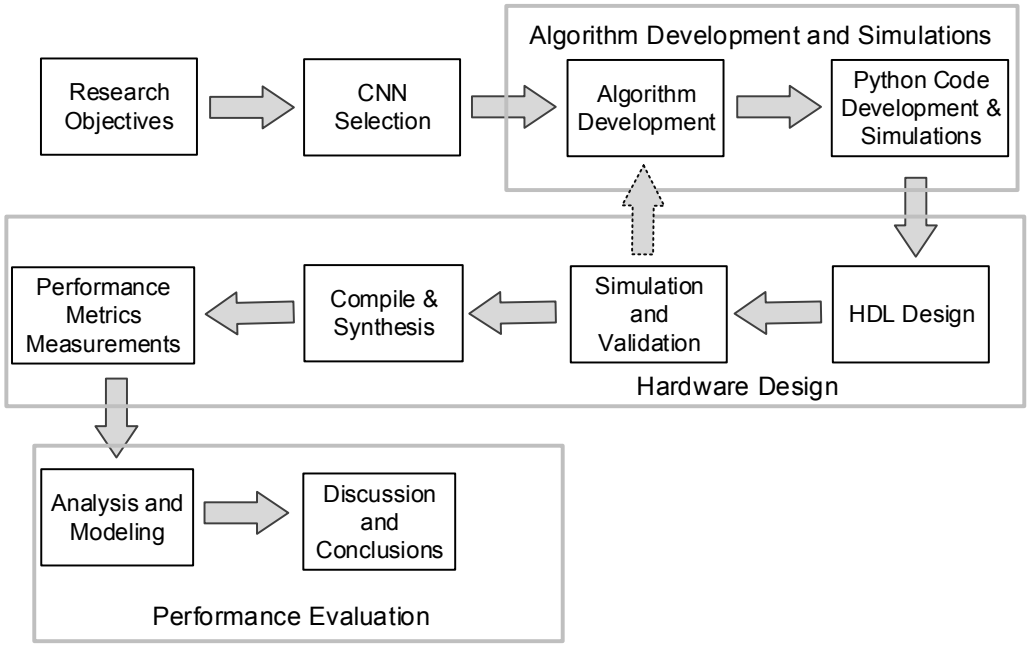

Figure 1. Research flow

3.2. Selection of CNN

The selection of a CNN architecture is a necessary step for conducting experiments and validating the design and implementation of an algorithm. Numerous CNNs have been proposed in the literature, with varying levels of complexity ranging from small models to deep architectures [55]. In this work, the following criteria are considered desirable for the selection of the CNN model:

- The model must fit in the FPGA device used in this research (i.e., Altera® Stratix® IV), and hence should be a small (i.e., lightweight) model;
- The model should be known to the research community.

We decided to select the 5-layers LeNet captured in Figure 2, which is designed for MNIST handwritten digits recognition [51] since it is simple, lightweight, and considered a representative CNN [1]. Also, LeNet CNN is well known and studied by numerous researchers in many studies such as the ones in [56–58].

LeNet consists of convolutional, pooling (i.e., subsampling), and fully connected layers. It accepts 32×32 digit image and produces ten outputs to classify the input digit. Table 1 describes the layers shown in Figure 2. The third column in Table 1 displays the feature map size, which is the size of the layer output. The feature map size is expressed as: width (W) $\times$ length (L) $\times$ depth (D) for a convolutional layer output and as depth (D) for an FC layer. The last column lists the number of parameters (weights and biases) required for the layer computations. The total number of parameters for the CNN model is 61,470 parameters. There are minor variations of LeNet in the literature (e.g., in terms of the activation functions). This research utilizes activation functions in the following way:

- The rectified linear unit (ReLU) activation function in C1 and C2;
- The sigmoid activation function in C3 and FC1;
- Softmax in the output layer.

For the third layer (i.e., C3), the size of its input feature map is $W \times L \times D = 5 \times 5 \times 16$, and its filter size is $W \times L$ (i.e., 5×5). In this case, the convolutional layer converges to an FC layer. This explains why this layer appears in both the convolutional and the FC entry (i.e., C3/FC0) in the 7th row in Table 1. While LeNet is referred to as 5 layers, some researchers consider Pooling (and even non-linear functions) as a separate layer. If Pooling

is counted as a separate layer, then LeNet is considered 7 layers [2]. Table 1 follows the latter convention.

Table 1. Layers of LeNet-5.

Layer	Description	Feature Map Size ($W \times L \times D$)	Filter Size	Stride	Parameters (Weights and Biases)
Input	input image	$32 \times 32 \times 1$	-	-	-
C1	convolutional	$28 \times 28 \times 6$	5×5	1	156
S1	pooling	$14 \times 14 \times 6$	2×2	2	
C2	convolutional	$10 \times 10 \times 16$	5×5	1	2416
S2	pooling	$5 \times 5 \times 16$	2×2	2	
C3/FC0	convolutional layer	$1 \times 1 \times 120$	5×5	1	48,120
FC1	Fully connected	84	-	-	10,164
FC2 (Output)	Fully connected	10	-	-	850

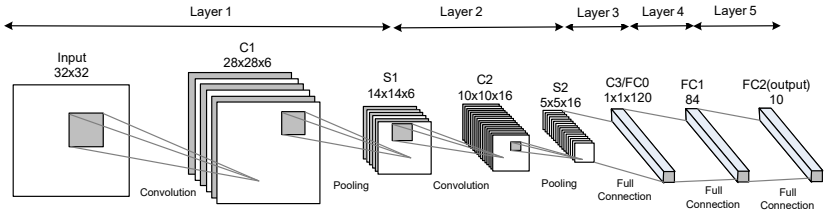

Figure 2. The architecture of LeNet-5

3.3. Algorithm Development and Simulations

This step encompasses the entire process from scratch, including design, debugging, and simulation of the proposed algorithm. Notably, we opted for a completely custom-designed CNN architecture (rather than utilizing pre-built libraries). This approach grants us maximum control over the network's structure, enabling a more thorough debugging process. There are several programs written in Python to facilitate this step which are as follows.

- The primary program implements the LeNet model and is capable of quantizing weights and/or activations of any layer with the desired bit width. This program is trained and tested with grayscale images of size 28×28 for the 10 digits (i.e., 0,1,. . . ,9). The training and testing data are chosen from the MNIST dataset [59]. The training parameters and hyper-parameters are listed in Table 2. One important constraint for our algorithm, which is discussed in Section 4, is to maintain the original architecture of the LeNet, which means it is not possible to add layers or neurons to the quantized CNN.
- Another program was written to generate the files containing quantized weights to be used in the hardware design. The weights must be expressed in binary or hexadecimal format.
- To debug and verify the hardware design, one more program was written to compare the model layer outputs with the hardware results and to determine the mismatching layer.

Table 2. Training parameters and hyper-parameters.

Name	Value
Dataset	MNIST [59]
Batch Size	50

Table 2. *Cont.*

Name	Value
Epochs	200
Optimizer	ADAM
Learning Rate	0.05
Loss Function	Cross Entropy

3.4. Hardware Design

In this research, the quantized CNN is implemented in hardware using a hardware description language (HDL), which is Verilog™. Our HDL design includes the CNN HDL implementation and the testbench to validate the results. The simulations are performed using the ModelSim simulator [60]. The HDL design incorporates the quantized parameters computed by our algorithm, which is verified at the end based on the testbench simulations.

During the initial design phase, findings from HDL design feasibility studies and simulation trials are used to refine specific algorithm parameters (detailed in Section 4). Figure 1 illustrates this feedback loop with the dotted arrow from the HDL simulation back to the algorithm development. This iterative process provides valuable insights from hardware design to improve the algorithm development.

Once the design is verified, the HDL is ready to be processed by the FPGA software tools. In this research, the FPGA that we used is the Altera Stratix IV [61], and we used the Quartus Prime™ software tool to process the HDL. Table 3 provides the versions of the FPGA device, tools, and simulator. Technical details regarding the software tool are available in [62]. Then, the HDL design goes through a flow similar to the flows discussed in published research, such as [63,64]. Finally, the compiled design is being analyzed and examined using the following performance metrics, which are thoroughly discussed next: resource utilization, timing, throughput, power, and energy.

Table 3. Field Programmable Gate Array (FPGA) device and tool versions.

Name	Value
FPGA Device	Altera Stratix IV version EP4SGX230
FPGA Software Tool	Quartus Prime Standard Edition version 17.0
Verilog Simulator	ModelSim-Intel FPGA Standard Edition version 10.5b

3.4.1. Resource Utilization Performance Metric

The resource utilization analysis lists the resources allocated for the design. The resources can be: Look-Up Table (LUT) and DSP blocks. An LUT is the smallest logical construct, which can be configured as a combinational logic or a register. Some versions of Altera FPGAs use Adaptive LUT (ALUT), which is an advanced version of the LUT and have the same number of inputs or outputs. From the documentation, there is no distinction in size between LUTs and ALUTs [62].

The DSP block implements an n-bit multiplier: $P = A \times B$. Optionally, input A or input B data are saved in an n-bit register; and the output P result is saved in a $2n$-bit register. To better compare different designs, we have chosen to approximate the resource utilization in terms of the LUTs. Consequently, an n-bit multiplier is approximated by LUTs as was demonstrated in [65] and based on Equation (1):

$$N_m \approx 1.25 \times n^2 \tag{1}$$

where N_m represents an n-bit multiplier size expressed in number of LUTs.

An n-bit register size is approximated by Equation (2):

$$N_r \approx n \tag{2}$$

where N_r represents an n-bit register size expressed in number of LUTs.

3.4.2. Timing and Throughput Performance Metrics

The timing analysis computes the design frequency by calculating the longest timing paths. Several trials for the timing runs identified 25 ns as the clock period (i.e., our timing constraint) that ensures reliable operation on the targeted FPGA while avoiding excessive resource consumption during synthesis. Additionally, the timing results are used to compute the implementation throughput as discussed in Section 6.2.

3.4.3. Power and Energy Performance Metrics

The power analysis, using the Power Analyzer^{TM} tool [62], computes the power dissipation of our implementation. The power is computed based on the resources, routing information, and node activity. The node activity is extracted from the waveform files (i.e., value-change dump files), which are produced by the ModelSim during simulating the design. Power Analyzer^{TM} reports core dynamic power (in mW), which includes four components: DSP block, combinational, register, and clock. Finally, the energy consumed during the processing of an image is the product of the power consumption and the processing time per image.

3.5. Performance Evaluation

During this step, we analyze the results to gain insights into CNN performance and draw key conclusions. This involves parsing and summarizing reports generated by FPGA tools for each performance metric, extracting crucial data points and trends. These summaries not only reveal data behavior but also serve as a foundation for building mathematical models of CNN performance on the target FPGA.

It is important to note that meaningful comparisons with prior work or baselines require implementing the same NN model on the same FPGA device. Comparisons across different models or devices can be misleading due to inherent variations. Therefore, in this research, where feasible quantized designs are limited by FPGA capacity and desired accuracy, we focus on comparing the performance of our implemented CNN designs against each other.

4. The Proposed Algorithm

Figure 3 illustrates the main steps of the proposed algorithm. This algorithm achieves full model quantization without introducing additional neurons. It leverages a hybrid approach, combining quantization training and post-training quantization. The process consists of two phases:

- Phase-I: Quantization training, which quantizes weights through dedicated training procedures.
- Phase-II: Post-training quantization, which further refines the model by optimally quantizing activations. No retraining is performed after the quantization because our experiments showed that it offered few advantages and could even have drawbacks.

Table 4 explains the algorithm parameters used in the algorithm discussion. Several parameters have recommended values or defined limits, which were established through extensive trial runs and feasibility simulations.

An N-bit fixed point has I integer bits and F fraction bits, where $N = I + F$. For example, the fixed-point signed number $Z = 011.10011$ has $I = 3$, $F = 5$, and $N = 8$. The N-bit signed value of Z is expressed by Equation (3):

$$VALUE_{I.F} = \frac{\text{Binary value of } N\text{-bit number without radix point}}{2^F} \tag{3}$$

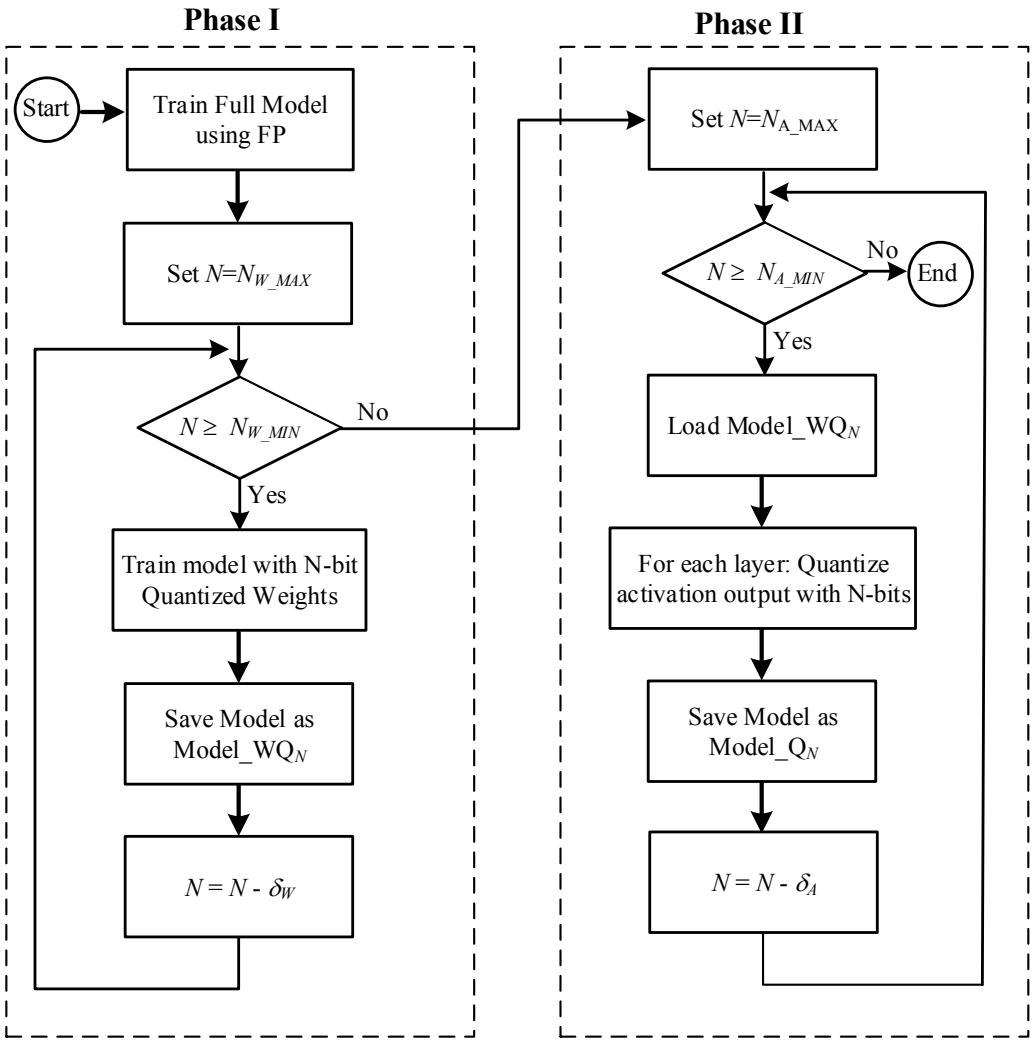

Figure 3. Algorithm flow.

So, the signed value of Z is equal to $\frac{||01110011||}{2^5} = \frac{+115}{32} = +3.59375$. The maximum and minimum values that can be represented by an N-bit number are expressed in Equations (4) and (5):

$$MAX\_VALUE_{I.F} = \frac{2^{(N-1)} - 1}{2^F} = 2^{(I-1)} - \frac{1}{2^F} \tag{4}$$

$$MIN\_VALUE_{I.F} = \frac{-2^{(N-1)}}{2^F} = -2^{(I-1)} \tag{5}$$

For example, an 8-bit representation with $I = 3$, $F = 5$ has a $MAX\_VALUE_{3.5} = 3.96875$ and a $MIN\_VALUE_{3.5} = -4.0$.

Table 4. Algorithm parameters.

Parameter	Description
N	Number of bits of the quantized number, which consists of an integer part and a fraction part.
I	Number of bits in the integer part in the quantized number
F	Number of bits in the fraction part of the quantized number

Table 4. *Cont.*

Parameter	Description
$VALUE_{I.F}$	The signed value of the N-bit fixed point number, with I-bit representing integer part and F-bit representing the fraction part.
$MAX\_VALUE_{I.F}$	The maximum signed value of an N-bit fixed point number, with I bits representing integer part and F-bit representing the fraction part.
$MIN\_VALUE_{I.F}$	The minimum signed value of an N-bit fixed point number, with I-bit representing integer part and F-bit representing the fraction part.
N_{W_MAX}	The initial number of bits to quantize weights; used in Phase-I of the algorithm. It is recommended that N_{W_MAX} = 32-bit, which is the integer size.
N_{W_MIN}	The final number of bits to quantize weights; used in Phase-I of the algorithm. To ensure the results accuracy, we set N_{W_MIN} = 8.
δ_W	The amount by which the number of quantized bits is reduced in each iteration of Phase-I. The typical value for δ_W = 2 to decrement both I and F by one.
Model_WQ$_N$	The CNN model with N-bit quantized weights.
N_{A_MAX}	The initial number of bits to quantize activations; used in Phase-II of the algorithm. Typically, $N_{A\_MAX} \leq N_{W\_MAX}$.
N_{A_MIN}	The final number of bits to quantize activations; used in Phase-II of the algorithm. To ensure the results accuracy, $N_{A\_MIN} = 8$.
δ_A	The amount by which the number of quantized bits is reduced in each iteration of Phase-II. Typical value for $\delta_A = 4$.
Model_Q$_N$	The CNN model with N-bit quantized weights and activations.

The algorithm phases, shown in Figure 3, are discussed in the following subsections.

4.1. Phase-I: Quantizing Weights

This phase utilizes a progressive and iterative quantization scheme for modeling the weights. This approach aims to achieve a balance between accuracy preservation and efficient storage/computation by quantizing the weights in a step-by-step manner. Initially, weights are quantized to $N = N_{W\_MAX}$. In each iteration, the model is trained with the quantized weights. At the end of the iteration, N is decremented by δ_W to start a new iteration.

We now present a breakdown of the steps within this phase (as illustrated in Figure 3):

1. Model is initially trained with a single precision floating point.
2. Set $N = N_{W\_MAX}$.
3. Quantize weights (and biases) to N-bit values, where $I = F = \frac{N}{2}$.
4. Train the model:
   - Perform forward pass with N-bit quantization on weights and biases.
   - Perform back-propagation with floating points for the entire model.
5. Save quantized weights (and biases) as Model_WQ$_N$.
6. Decrement N by δ_W.
7. Check if iterations are completed:
   - If $N \geq N_{W\_MIN}$: Go to step 3.
   - Otherwise: terminate Phase-I.

4.2. Phase-II Quantizing Activations

Building upon Phase-I's quantized weights, Phase-II refines the model by quantizing activations for specific bit-widths (N). However, not all quantized models from Phase-I are utilized, resulting in a subset of N values explored in Phase-II. For activation quantization, we employ an optimized technique called "funnel bit assignment". The output

bits' selection in this approach works similarly to that of a funnel shifter design [66]. This optimization leverages a set of the input images, known as "regression images", to guide the quantization process, ensuring optimal bit allocation.
The following are the steps of Phase-II (as illustrated in Figure 3):

1. Set $N = N_{A\_MAX}$.
2. Load Model_WQ$_N$, which is generated by Phase-I.
3. For the weights that have small integer part (i.e., integer part < 2^{I-1}), assign more bits to the fractional part to achieve better accuracy.
4. Run regression images through the model.
5. For each layer$_i$, $i \in 1, \ldots, 7$, perform funnel bit assignment:
   (a) Compute the maximum and minimum values of activation outputs.
   (b) Determine the number of bits I required to store the integer part. This is done by computing the number of bits to store the maximum and minimum values of activations: I_1, I_2.
      - $I_1 = log_2(\text{Max Activation Value})$.
      - $I_2 = log_2(abs(\text{Min Activation Value}))$.
      - Set I to the larger value of I_1 and I_2.
   (c) Perform bit assignment:
      - If $I < N$: assign I bits to the integer part and F bits to the fraction part, where $F = N - I$.
      - If $I \geq N$: assign all N bits to upper bits of the integer part and no bits are assigned to the fraction part. Effectively, $I = N$ and $F = 0$.
   (d) Run regression images and record accuracy. When computing activation output:
      - If an activation is above the $MAX\_VALUE_{I.F}$, then saturate the output to $MAX\_VALUE_{I.F}$.
      - If an activation is below the $MIN\_VALUE_{I.F}$, then saturate the output to $MIN\_VALUE_{I.F}$.
   (e) Decrement I by one, repeat steps (c)–(e) to find out the optimum assignment.
6. Save model as Model_Q$_N$.
7. Decrement N by δ_A.
8. Check if iterations are completed:
   - If $N \geq N_{A\_MIN}$: Go to step 2.
   - Otherwise: Terminate Phase-II.

5. Hardware Design

In this section, we discuss the hardware implementation of the LeNet model, which is shown in Figure 4. The implementation is scalable and can be fine-tuned to any other CNN's. The hardware design consists of the following main units:

- Layer units realize the model layers corresponding to Figure 2 and Table 1. The units perform the computations of the LeNet model. They include:
  - CONV 1, CONV 2, and CONV 3, which implement the convolutional layers C1, C2, and C3, respectively.
  - POOL 1 and POOL 2, which implement the S1 and S2 layers, respectively.
  - FC 1 and FC 2 are the fully connected layer implementations.
- Memory units implement the data memory and the interfacing logic. The interface logic enables continuous data access by using two data memories, as explained below. The units include:
  - Memory interface (Mem I/F), which is a dedicated unit to facilitate communication between the CNN model and the memory system.
  - Two data memories, Mem$_A$ and Mem$_B$, which store the model's input data and activation values.

– ROMs to store the weights and biases used by the model.

The following subsections present a detailed description of the hardware design and its configurability.

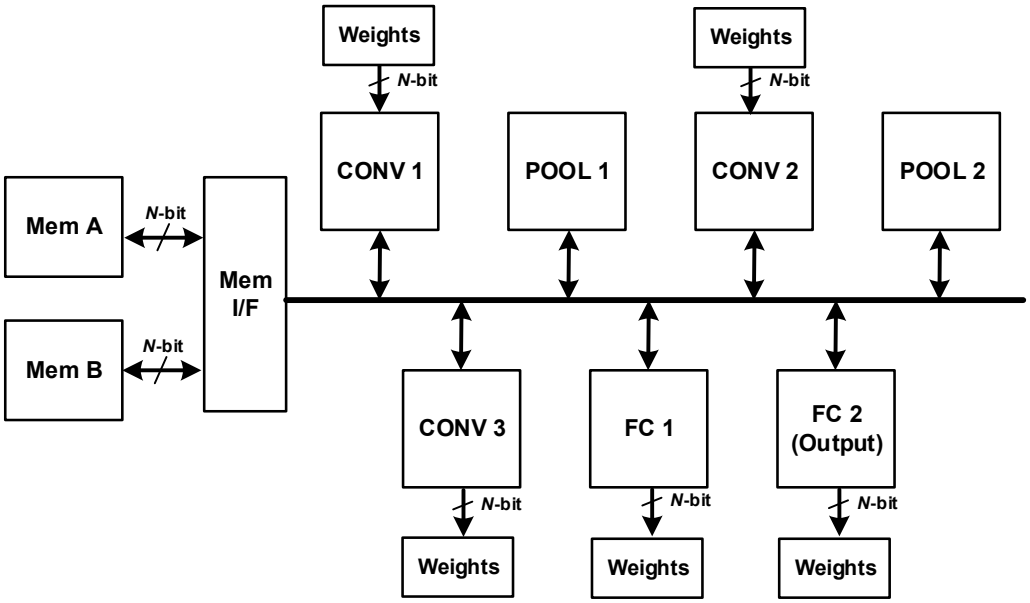

Figure 4. Convolutional neural networks (CNN) hardware design.

5.1. Design Configurability

Early in the design process, we faced an important issue: how to efficiently design multiple similar units like CONV 1, CONV 2, and CONV 3. Two options were examined: independent units versus configurable units. The option of independent units requires designing each CONV layer as a separate hardware unit. This approach requires individual design and verification efforts for each unit, leading to potentially higher development time. On the other hand, the configurable unit designs a single, configurable unit capable of executing the functionality of all three CONV layers. This option requires careful design for configuration and flexibility but allows for significant design reuse, which reduces development time and effort. We ultimately chose the second option, the configurable unit, to leverage the benefits of design reuse.

Furthermore, to accommodate diverse *N*-bit implementations, the design incorporates configurability within datapath units, memories, and buses. This allows for hardware customization based on specific bit-width requirements.

5.2. Layer Units

The design of CONV units (i.e., CONV 1, CONV 2, and CONV 3), which is illustrated in Figure 5 and consists of a control unit and datapath unit. The control unit consists of the finite-state machine (FSM) to manage the activities. It handles the handshakes with other units, generates the control signals for the datapath blocks, and computes addresses for ROM and MEM I/F. The datapath unit consists of a Multiply-Add block, activation block, weight ROM, multiplexer, and register.

The Multiply-Add block computes one column of the 5×5 convolution operation, and hence, 5 cycles are required to complete the convolution. The activation function block performs the non-linear activation operation. The multiplexer and register send the registered partial results back to the Multiply-Add block or send the final result to the MEM/IF. The ROM and Mem I/F supply the Multiply-Add block with data and weights/biases.

The initial design exploration examined the ROM implementation options, which are the memory versus constant tables as synthesized by the FPGA tool. Ultimately, it was

decided to use the tables due to energy concerns in the RCD (i.e., FPGA) that we considered in this research. The relatively small size of the ROMs and memory energy consumption played key roles in favoring the constant tables. In general, implementing small-sized ROMs using LUT is more energy-efficient compared with RAMs [67].

The adder block is carefully designed to select the appropriate N-bits out of the addition operation result. This is to support the shifting operation required by the algorithm in Phase-II. The selection is controlled by the shift signal generated by the control unit. For example, if shift = 0, then the selected N-bits are the most significant N-bit of the addition result. If shift = 1, then the selected N-bits are those to the right of the most significant bit of the addition result. Furthermore, to avoid overflow, adder output is saturated to $MAX\_VALUE_{I.F}$ or $MIN\_VALUE_{I.F}$ values in the following cases:

- If the addition result > $MAX\_VALUE_{I.F}$, then the output is set to $MAX\_VALUE_{I.F}$.
- If the addition result < $MIN\_VALUE_{I.F}$, then the output is set to $MIN\_VALUE_{I.F}$.

"POOL" and "FC" units have similar overall structure with differences in the implementation details.

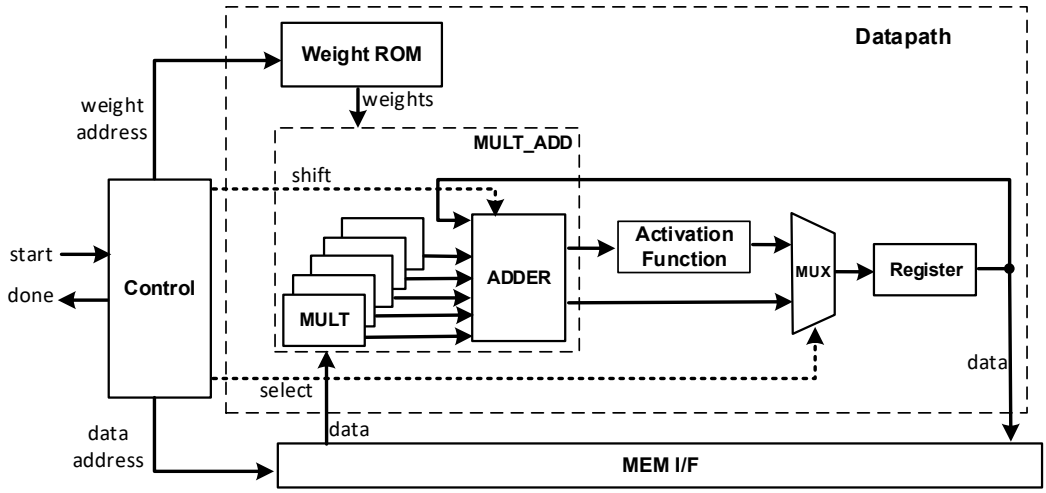

Figure 5. Convolutional layer hardware design.

5.3. Memory Units

Figure 6 illustrates the block diagram of the memory interface and data memory. To optimize the design performance, the system leverages two distinct memories (MEM-A and MEM-B) and operates in two dedicated modes, maximizing efficiency and resource utilization. When mode = 0, the design reads from MEM-A and writes to MEM-B; when mode = 1, the design reads from MEM-B and writes to MEM-A. This organization allows the design to perform read/write operations without any interruption or delays. At the start of processing an image, the image is loaded in MEM-A. CONV 1 layer executes with mode = 0; it reads from MEM-A and writes to MEM-B. Then, POOL 1 executes with mode = 1; it reads from MEM-B and writes to MEM-A. This continues for the rest of the layers.

5.4. Execution Flow

The execution of the hardware operations in CNN hardware starts by asserting the "start" signal of CONV 1 unit. When CONV 1 completes its operations, it asserts the "done" signal, which triggers the "start" signal of POOL 1. This continues with the rest of the units, until FC 2 asserts "done", which is the end of the CNN hardware operations.

Inside each unit, when the "start" signal arrives, it informs the unit to begin processing the data. In each clock, the addresses of data and weights are dispatched to the MEM I/Fs and the ROMs. When data and weight arrive, the Multiply-Add block performs multiplication operations and adds the results.

If the convolution operation is complete, the result is processed by the activation function and saved in the register. The next cycle, data, and its address are dispatched to the MEM I/F. If the convolution operation is in progress, the partial result is saved in the register. In the next cycle, the partial result is sent as input to the addder.

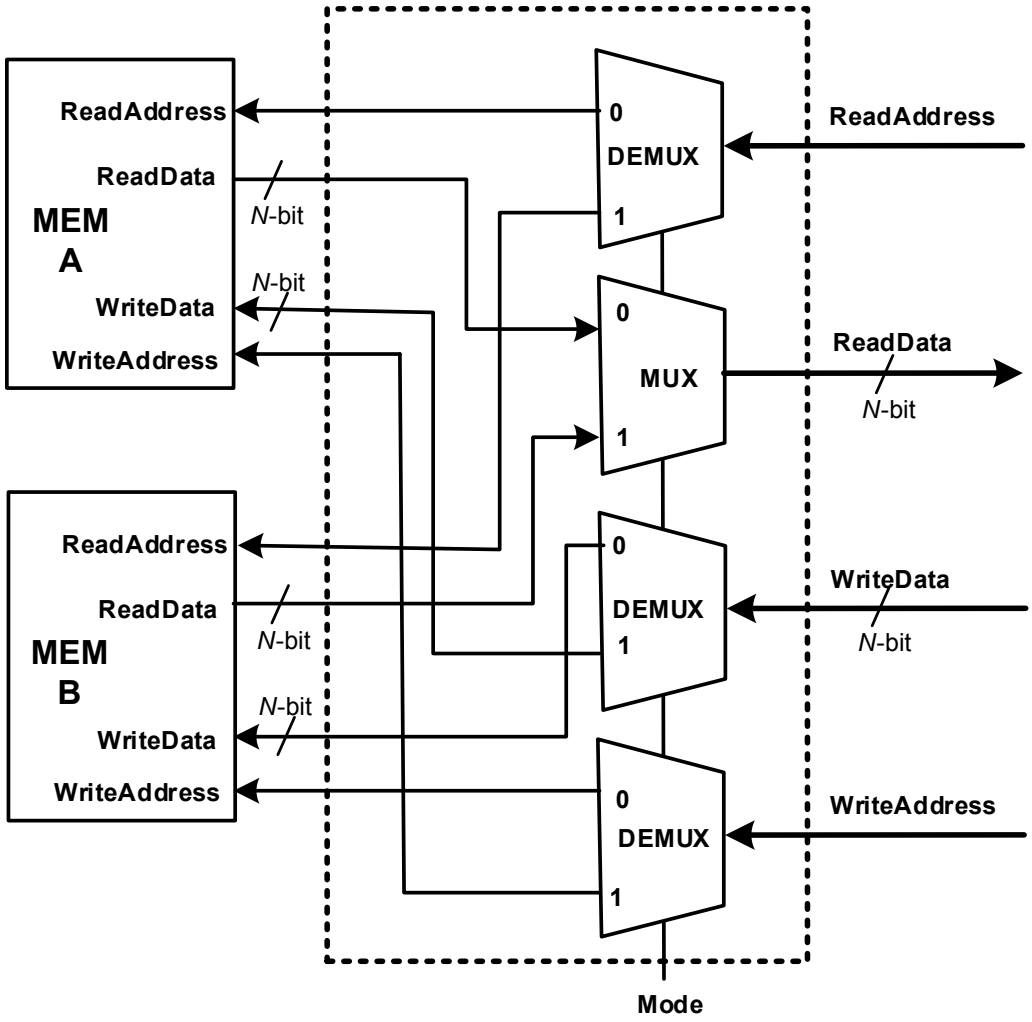

Figure 6. Memory interface and data memory.

6. Implementation Results

As discussed in Section 3, the HDL design is processed by the FPGA tools to compile, synthesize, and fit the design on the FPGA. The CNN model was quantized for $N = 32$-, 24-, 20-, 16-, 12-, and 8-bit. All quantized models have 1-top accuracy > 98.5%. During the fitting phase, the design with $N = 32$- , 24-, and 20-bit did not fit in the FPGA device due to insufficient resources (this can be inferred from the presented results in the last column of Table 5 as shall be discussed in the following subsection). This truly presented us with a real challenge: to fit the CNN model in an RCD, which is the FPGA in this case. However, we succeeded in fitting the FPGA with $N = 16$-, 12-, and 8-bit. The rest of the section provides a summary of the performance metrics.

6.1. Resource Utilization

Table 5 summarizes the resource utilization, as discussed below:

- The second and third columns list the LUTs and register utilization.
- The fourth column to the seventh columns list the 9-bit, 12-bit, 18-bit, and 36-bit multipliers used in the design.

- The eighth column computes the total LUTs, with multipliers estimated as discussed in Section 3.
- The ninth column normalizes the results with respect to the 16-bit design.
- The last column lists the logic utilization, which is calculated as the ratio of the used resources to the total available resources. Looking at the trend of the logic utilization versus N in this column clearly shows that it is not possible to fit more than a 16-bit design in the FPGA.

It should be noted that reducing $N = 16$ to $N = 12$ (and also reducing $N = 12$ to $N = 8$), reduces resources by about 26%. However, reducing N by half (i.e., reducing $N = 16$ to $N = 8$), reduces the design resources by 46%. This means that the design exhibits excellent scalability with respect to N, meaning most of the components or resources reduce proportionally to N with only a minor fixed overhead logic. The overhead logic includes control circuits, which do not scale with N.

Table 5. Resource utilization for 16-bit, 12-bit, and 8-bit quantized models.

Des.	LUTs	Reg. Utilization	9-b M.	12-b M.	18-b M.	36-b M.	Tot. LUT	Norm. Res.	Logic Util.
16-bit	76,217	821	0	0	20	5	93,778	1	82%
12-bit	60,683	741	0	20	0	5	73,544	0.78	65%
8-bit	45,866	659	20	0	5	0	50,840	0.54	49%

6.2. Timing Analysis

Timing analysis is performed based on the timing constraints presented in Section 3. Table 6 shows the reported maximum frequencies for different values of N. The last column in the table normalizes the frequencies with respect to the 16-bit design. The values in Table 6 reveal a trend: the maximum frequency tends to be slightly higher for smaller values of N. For instance, the maximum frequency for 8-bit is roughly 2% higher than the one for 16-bit. This observation might be attributed to the efficient scalability of our design, which is a result of its well-considered parametric structure. We should mention that adjusting a few parameters in the HDL code allows for adaptation to various N-bit designs, with the primary design changes occurring in the datapaths (longest timing paths). These alterations leverage pre-designed and optimized datapath units within the FPGA, leading to minimal timing variations (few logic levels of delay) across different designs. Additionally, the timing analysis helps us compute the throughput (i.e., processed images per second) using Equation (6):

$$\begin{aligned} Throughput(Image/Second) &= \frac{1}{Time_{ProcessOneImage}} \\ Time_{ProcessOneImage} &= Cycles_{ProcessOneImage} \times CycleTime \end{aligned} \tag{6}$$

where:

- $Time_{ProcessOneImage}$ is the time required to process one image.
- $Cycles_{ProcessOneImage}$ represents the total number of clock cycles needed to process a single image.
- $CycleTime$ refers to the duration of one clock cycle.

The throughput results of the 16-, 12-, and 8-bit designs are 328 image/second, 333 image/second, and 334 image/second, respectively.

Table 6. Maximum frequency (MHz) for 16-bit, 12-bit, and 8-bit quantized models.

Design	Fmax	Norm. Fmax
16-bit	43.33	1.000

Table 6. *Cont.*

Design	Fmax	Norm. Fmax
12-bit	44.06	1.017
8-bit	44.11	1.018

6.3. Power and Energy Consumption

Table 7 summarizes the power results for our implemented designs of varying N-bit, where $N \in (16, 12, 8)$. Columns two through five show the power dissipation for combinational cells, registers, clock, and DSP blocks, and columns six and seven list the total power and normalized total power with respect to the 16-bit design, respectively.

Looking carefully at Table 7, the following can be inferred.

- The combined power consumption of the clock and registers, known as sequential power, accounts for only 2% of the total power, playing a minor role in overall power consumption. This is primarily because these sequential circuits occupy a smaller area compared to the more power-hungry data path logic.
- The power consumption of the DSP blocks is around 5% of the total power. This indicates that the datapath resources like multipliers and adders are well-designed for low power.
- The combinational cells are the primary source of power consumption, accounting for approximately 93% of the total power. This is primarily due to the presence of random logic, high routing overhead, and large data selection components like multiplexers and demultiplexers.
- The final column of Table 7 showcases the superior power saving of the 8-bit design, consuming around 41% less power compared to the 16-bit design. While the 12-bit design offers a modest 2% power saving over the 16-bit option. This unexpectedly low power saving in the 12-bit design prompted further investigation. We believe the root cause might be the combined nature of combinational cell power: it includes both block power and routing power. The routing power is the power consumed by the metal wires and the routing resources that connect the logic blocks. It increases with the wire length and complexity of routing paths. Interestingly, the 12-bit design displayed minimal change, even a slight increase in routing power compared to the 16-bit design. We suspect this anomaly might indicate an issue with the software tool's routing algorithm, potentially favoring byte-aligned sizes (8-bit and 16-bit) and hindering efficiency for non-aligned designs like the 12-bit one. While other factors could contribute to the 12-bit power consumption, Table 7 clearly demonstrates the scalability of power consumption with a 41% reduction achieved by halving the bit-width from 16 to 8. This power saving is achieved by 41% power reduction in the combinational cell power and 35% reduction in the DSP power.

Table 7. Power results (mW) for 16-bit, 12-bit, and 8-bit quantized models.

Design	Comb. Cells	Registers	Clock	DSP Power	Total Power	Norm. Total Power
16-bit	857.77	3.79	9.94	52.84	924.34	1.0
12-bit	855.56	2.26	9.04	37.97	904.83	0.98
8-bit	501.26	3.23	10.32	34.61	549.42	0.59

Now, regarding the energy analysis, it is captured by Equation (7), which shows how the energy of processing one input (i.e., an image) is being computed:

$$Energy = Power \times Time_{ProcessOneImage} \quad (7)$$

The analyzed energy results, based on Equation (7) and presented in Table 8 convincingly demonstrate the design's energy scalability. By simply halving the bit-width from 16 to 8 bits, we achieved a significant 42% reduction in energy consumption as shown in the last column of Table 8. However, the 12-bit design's energy saving remains modest due to the previously discussed power anomaly.

Table 8. Energy results (mJ) for 16-bit, 12-bit, and 8-bit quantized models.

Design	Total Energy	Norm. Energy
16-bit	2.82	1.0
12-bit	2.72	0.96
8-bit	1.65	0.58

7. Discussion

While Section 6 comprehensively analyzed and discussed the results of designs with varying bit widths (N), this section delves into their broader implications in terms of modeling and implementation within DNNs.

7.1. Modeling Performance Metrics

It is important to model the behavior of the performance metrics in terms of N-bit quantization. We applied linear modeling to the data. We then assessed the model's "goodness of fit" by calculating the coefficient of determination, also known as R-Squared (R^2).

The total LUT and logic utilization can be modeled by Equation (8) with $R^2 = 0.99$. This means that reducing the number of quantization bits by 1 is expected to save approximately 5.4 K LUTs and decrease logic utilization by 4%.

$$\begin{aligned} Total\_LUT &= 5367 \times N + 8314 \\ Logic\_Utilization\ (\%) &= 4.13 \times N + 15.83 \end{aligned} \tag{8}$$

Examining the power and energy results in Tables 7 and 8, clearly these metrics exhibit slight non-linearity. However, we believe a linear model provides a good fit to the data and offers valuable simplicity for system design. Therefore, we have chosen to model power and energy linearly in this work. Equation (9) models power with $R^2 = 0.79$. The reduced value of R^2 is due to the 12-bit power, which was considered in deriving the model. This shows clearly that reducing 1-bit in quantization saves around 46.9 mW in power.

$$Power\ (\mathrm{mW}) = 46.9 \times N + 230.5 \tag{9}$$

Finally, Equation (10) shows an energy model with $R^2 = 0.82$. This means a reduction of one quantization bit should save around 147 µJ.

$$Energy\ (\mu\mathrm{J}) = 146.7 \times N + 634 \tag{10}$$

7.2. Applying the Proposed Algorithm to DNN

So far, we have demonstrated the effectiveness of our proposed quantization algorithm on the LeNet CNN model. However, the scalability and ability to model various metrics achieved through our two-phase approach suggest its broader applicability to deeper neural networks (DNNs). While hardware implementation is beyond the scope of this work, the following analysis serves as an illustrative example showcasing the potential application and comprehensiveness of our concepts and models to DNNs. This paves the way for future hardware implementations and offers a valuable foundation for further exploration.

We assume that a DNN is implemented using multiple FPGAs. A group of DNN layers (referred to them as a single partition) are fitted in a single FPGA. A partition has a number of layers equivalent to those in LeNet, and it is denoted as NL_P. Similar to LeNet,

we set $\mathrm{NL}_P = 7$. If we assume that a DNN is partitioned to M partitions, then the number of layers in the DNN is expressed in Equation (11):

$$\mathrm{NL}_{DNN} = M \times \mathrm{NL}_P \tag{11}$$

We also assume the following (see Figure 7):

- $M = 3$. This number is chosen as a rough estimate, and it can easily be changed, and it means that we need to partition the design into three partitions, each fits in an FPGA device, where $\mathrm{Partition}_i$ is quantized with N_i-bit as captured in Figure 7.
- The complexity of each DNN layer is the same as those of the LeNet Model, otherwise, the models should be scaled up.
- The design is fitted on the same FPGA device utilized in this research. If another device is used, then update the models.
- The DNN model is quantized using the same N-bit values as we used before (i.e., Each N_i is chosen in the range from 16-bit to 8-bit).

Figure 7 shows the M partitions that we assumed above (i.e., $M = 3$), where each partition is modeled on an FPGA device, and has its own N_i-bit quantization as represented by N_1, N_2, and N_3, respectively, for the three layers.

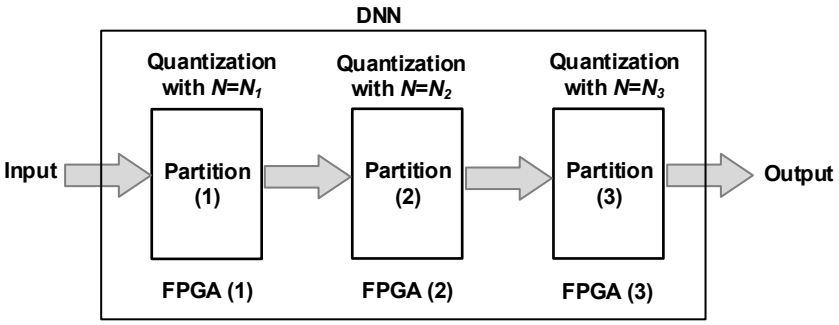

Figure 7. Deep neural networks (DNN) partitions.

Given the DNN's size (as controlled by the M parameter), varying quantization across partitions is possible. To maintain accuracy, it is generally recommended to use higher precision (larger quantization levels, N_i) for earlier layers in the network i.e., $N_1 \geq N_2 \geq N_3$. This ensures that errors introduced by lower precision (smaller N_i) in the initial stages do not propagate and accumulate throughout the network, potentially leading to incorrect final outputs. Several research works, such as [68], have demonstrated the effectiveness of using higher precision quantization for early layers as a design choice. Thus, we suggest quantizing starts from $\mathrm{Partition}_1$ to $\mathrm{Partition}_M$. For a $\mathrm{Partition}_i$, we perform the following steps:

- Run Phase-I of the algorithm on $\mathrm{Partition}_i$ to $\mathrm{Partition}_M$;
- Run Phase-II of the algorithm to quantize $\mathrm{Partition}_i$ and select N_i;
- For the rest of the steps, $\mathrm{Partition}_i$ is quantized at N_i.

To illustrate those steps, we apply them on the above DNN:

- Quantize $\mathrm{Partition}_1$;
  - Run Phase-I of the algorithm on the entire DNN (i.e., $\mathrm{Partition}_1$, $\mathrm{Partition}_2$ and $\mathrm{Partition}_3$);
  - Run Phase-II of the algorithm to quantize $\mathrm{Partition}_1$ and select N_1;
  - For the rest of the steps, $\mathrm{Partition}_1$ is quantized at N_1.
- Quantize $\mathrm{Partition}_2$
  - Run Phase-I of the algorithm on $\mathrm{Partition}_2$ and $\mathrm{Partition}_3$;
  - Run Phase-II of the algorithm to quantize $\mathrm{Partition}_2$ and select N_2, where $N_2 \leq N_1$;
  - For the rest of the steps, $\mathrm{Partition}_2$ is quantized at N_2.

- Quantize Partition$_3$
  - Run Phase-I of the algorithm on Partition$_3$;
  - Run Phase-II of the algorithm to quantize Partition$_3$ and select N_3, where $N_3 \leq N_2$.

The total LUT's average logic utilization, power utilization, and energy (of the above suggested DNN) are modeled and expressed, following linear regression modeling, as in Equation (12), Equation (13), Equation (14), and Equation (15), respectively:

$$\text{Total\_LUT}_{DNN} = 8314 \times M + \Sigma_{i=1}^{M} N_i \times 5367 \quad (12)$$

$$\text{Average\_Logic\_ Utilization}_{DNN} = 15.83 + \frac{\Sigma_{i=1}^{M} N_i \times 4.13}{M} \quad (13)$$

$$\text{Power}_{DNN}(\text{mW}) = 230.5 \times M + \Sigma_{i=1}^{M} N_i \times 46.9 \quad (14)$$

$$\text{Energy}_{DNN}(\mu\text{J}) = 634 \times M + \Sigma_{i=1}^{M} N_i \times 146.7 \quad (15)$$

Assuming the above steps resulted in N_1 = 16-bit, N_2 = 12-bit, and N_3 = 8-bit for each of the three partitions, Table 9 displays the predicted performance metrics for the DNN model. These predicted metrics offer valuable insights into the DNN's resource requirements, power consumption, and energy usage. This information can significantly influence crucial design decisions. For instance, the predicted power consumption can inform the selection of a suitable power supply capable of meeting the DNN's average power needs. Similarly, the predicted energy usage can help determine if the system's battery can adequately handle the power demands of DNN operations. Furthermore, the total number of LUTs plays a role in determining the appropriate size of future FPGA device needed.

Table 9. Deep neural networks (DNN) performance metrics.

Total LUT	Ave. Logic Util. %	Power (W)	Energy (mJ)
218,154	65.4%	2.38	7.18

8. Conclusions and Future Works

In this research, we presented a scalable quantization algorithm to reduce the CNN hardware design and estimate and model the area, power, and energy. The algorithm combines quantization training and post-training quantization techniques and it provides full model quantization without increasing the model complexity.

We implemented LeNet in Altera Stratix® IV FPGA. The algorithm was applied to quantize the model to various bit widths. We succeeded in implementing the model on the FPGA using 16-, 12-, and 8-bit quantizations. Compared to the 16-bit design, the 8-bit design offers improved resource efficiency with a 44% decrease in LUT utilization, and it achieves power and energy reductions of 41% and 42%, respectively.

The derived models show significant value in estimating design metrics. This indicates that trading off one quantization bit yields savings of approximately 5.4 K LUTs, 4% logic utilization, 46.9 mW power, and 147 µJ energy. We also demonstrated the practical use of the derived models to estimate performance metrics for a sample DNN design.

Future efforts could investigate the expansion of the algorithm to handle aggressive quantization, such as 4-bit or even lower. Subsequently, modeling the metrics at these extremely low-precision levels would provide valuable insights into their effects on resources and performance. Moreover, exploring architectural options like pipelining could potentially lead to significant design savings. Also, exploring the feasibility and impact of merging computations from multiple layers into a unified layer deserves further research. Furthermore, future work can explore the impact of alternative quantization techniques and conduct a broader comparison study to optimize NN design efficiency. Finally, applying the algorithm to a recent, representative DNN architecture presents an exciting opportunity to assess its potential for substantial performance improvements. This practical evaluation could solidify the algorithm's effectiveness and real-world impact.

Author Contributions: Conceptualization, B.J.M., K.M.A.Y., A.A. and T.H.; methodology, B.J.M., K.M.A.Y., A.A. and T.H.; software, B.J.M., K.M.A.Y., A.A. and T.H.; validation, K.M.A.Y.; formal analysis, B.J.M., K.M.A.Y., A.A. and T.H.; investigation, B.J.M., K.M.A.Y., A.A. and T.H.; resources, B.J.M., K.M.A.Y., A.A. and T.H.; data curation, B.J.M., K.M.A.Y., A.A. and T.H.; writing—original draft preparation, B.J.M., K.M.A.Y., A.A. and T.H.; writing—review and editing, B.J.M., K.M.A.Y., A.A. and T.H.; visualization, B.J.M., K.M.A.Y., A.A. and T.H.; supervision, B.J.M.; project administration, B.J.M. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Data Availability Statement: The data presented in this study are available on request from the corresponding author.

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

Abbreviations

The following abbreviations are used in this manuscript:

CNN	Convolutional Neural Network
NN	Neural Network
DNN	Deep Neural network
RCD	Resource-Constrained Devices
FC	Fully Connected
SVD	Singular Value Decomposition
FPGA	Field-Programmable Gate Array
QAT	Quantization-Aware Training
PTQ	Post-Training Quantization
BNN	Binarized Neural Network
Q*n.m*	Fixed-point representation with *n* bits for the integer part, *m* bits for the fractional part
NN parameters	Weights/synapses and biases
Feature map	Output of convolutional layer
FSM	Finite State Machine
STE	Straight-Through Estimator
MNIST	Modified National Institute of Standards and Technology
TIMIT	Texas Instruments/Massachusetts Institute of Technology

References

1. Zhang, W. A Survey of Field Programmable Gate Array-Based Convolutional Neural Network Accelerators. *Int. J. Electron. Commun. Eng.* **2020**, *14*, 419–427.
2. Mittal, S. A survey of FPGA-based accelerators for convolutional neural networks. *Neural Comput. Appl.* **2020**, *32*, 1109–1139. [CrossRef]
3. Gholami, A.; Kim, S.; Dong, Z.; Yao, Z.; Mahoney, M.W.; Keutzer, K. A survey of quantization methods for efficient neural network inference. *arXiv* **2021**, arXiv:2103.13630.
4. Kim, Y.D.; Park, E.; Yoo, S.; Choi, T.; Yang, L.; Shin, D. Compression of deep convolutional neural networks for fast and low power mobile applications. *arXiv* **2015**, arXiv:1511.06530.
5. Qiu, J.; Wang, J.; Yao, S.; Guo, K.; Li, B.; Zhou, E.; Yu, J.; Tang, T.; Xu, N.; Song, S.; et al. Going deeper with embedded fpga platform for convolutional neural network. In Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, Monterey, CA, USA, 21–23 February 2016; pp. 26–35. [CrossRef]
6. Chen, T.; Du, Z.; Sun, N.; Wang, J.; Wu, C.; Chen, Y.; Temam, O. Diannao: A small-footprint high-throughput accelerator for ubiquitous machine-learning. *ACM SIGARCH Comput. Archit. News* **2014**, *42*, 269–284. [CrossRef]
7. Mohd, B.J.; Hayajneh, T.; Vasilakos, A.V. A survey on lightweight block ciphers for low-resource devices: Comparative study and open issues. *J. Netw. Comput. Appl.* **2015**, *58*, 73–93. [CrossRef]
8. Zhuang, B.; Shen, C.; Tan, M.; Liu, L.; Reid, I. Towards effective low-bitwidth convolutional neural networks. In Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, Salt Lake City, UT, USA, 18–22 June 2018; pp. 7920–7928.
9. Zhu, C.; Han, S.; Mao, H.; Dally, W.J. Trained ternary quantization. *arXiv* **2016**, arXiv:1612.01064.
10. Miyashita, D.; Lee, E.H.; Murmann, B. Convolutional neural networks using logarithmic data representation. *arXiv* **2016**, arXiv:1603.01025.
11. Chen, Y.H.; Krishna, T.; Emer, J.S.; Sze, V. Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks. *IEEE J. Solid-State Circuits* **2016**, *52*, 127–138. [CrossRef]

12. Ma, Y.; Suda, N.; Cao, Y.; Seo, J.s.; Vrudhula, S. Scalable and modularized RTL compilation of convolutional neural networks onto FPGA. In Proceedings of the 2016 26th International Conference on Field Programmable Logic and Applications (FPL), Lausanne, Switzerland, 29 August–2 September 2016; IEEE: Piscataway, NJ, USA, 2016; pp. 1–8.
13. Suda, N.; Chandra, V.; Dasika, G.; Mohanty, A.; Ma, Y.; Vrudhula, S.; Seo, J.s.; Cao, Y. Throughput-optimized OpenCL-based FPGA accelerator for large-scale convolutional neural networks. In Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, Monterey, CA, USA, 21–23 February 2016; pp. 16–25.
14. 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.
15. Novac, P.E.; Boukli Hacene, G.; Pegatoquet, A.; Miramond, B.; Gripon, V. Quantization and deployment of deep neural networks on microcontrollers. *Sensors* **2021**, *21*, 2984. [CrossRef] [PubMed]
16. Guo, K.; Zeng, S.; Yu, J.; Wang, Y.; Yang, H. [DL] A survey of FPGA-based neural network inference accelerators. *ACM Trans. Reconfig. Technol. Syst. (TRETS)* **2019**, *12*, 1–26. [CrossRef]
17. Abdelouahab, K.; Bourrasset, C.; Pelcat, M.; Berry, F.; Quinton, J.C.; Sérot, J. A Holistic Approach for Optimizing DSP Block Utilization of a CNN implementation on FPGA. In Proceedings of the 10th International Conference on Distributed Smart Camera, Paris, France, 12–15 September 2016; pp. 69–75.
18. Zhang, X.; Zou, J.; He, K.; Sun, J. Accelerating very deep convolutional networks for classification and detection. *IEEE Trans. Pattern Anal. Mach. Intell.* **2015**, *38*, 1943–1955. [CrossRef] [PubMed]
19. Novikov, A.; Podoprikhin, D.; Osokin, A.; Vetrov, D.P. Tensorizing neural networks. *Adv. Neural Inf. Process. Syst.* **2015**, *28*.
20. Szegedy, C.; Liu, W.; Jia, Y.; Sermanet, P.; Reed, S.; Anguelov, D.; Erhan, D.; Vanhoucke, V.; Rabinovich, A. Going deeper with convolutions. In Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, Boston, MA, USA, 7–12 June 2015; pp. 1–9.
21. Iandola, F.N.; Han, S.; Moskewicz, M.W.; Ashraf, K.; Dally, W.J.; Keutzer, K. SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and <0.5 MB model size. *arXiv* **2016**, arXiv:1602.07360.
22. He, K.; Zhang, X.; Ren, S.; Sun, J. Deep residual learning for image recognition. In Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, Las Vegas, NV, USA, 27–30 June 2016; pp. 770–778.
23. Zhang, X.; Zhou, X.; Lin, M.; Sun, J. Shufflenet: An extremely efficient convolutional neural network for mobile devices. In Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, Salt Lake City, UT, USA, 18–22 June 2018; pp. 6848–6856.
24. Chollet, F. Xception: Deep learning with depthwise separable convolutions. In Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, Honolulu, HI, USA, 21–26 July 2017; pp. 1251–1258.
25. Howard, A.G.; Zhu, M.; Chen, B.; Kalenichenko, D.; Wang, W.; Weyand, T.; Andreetto, M.; Adam, H. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *arXiv* **2017**, arXiv:1704.04861.
26. Huang, P.; Wu, H.; Yang, Y.; Daukantas, I.; Wu, M.; Zhang, Y.; Barrett, C. Towards Efficient Verification of Quantized Neural Networks. In Proceedings of the AAAI Conference on Artificial Intelligence, Stanford, CA, USA, 25–27 March 2024; Volume 38, pp. 21152–21160.
27. Cheng, L.; Gu, Y.; Liu, Q.; Yang, L.; Liu, C.; Wang, Y. Advancements in Accelerating Deep Neural Network Inference on AIoT Devices: A Survey. *IEEE Trans. Sustain. Comput.* **2024**, 1–18. [CrossRef]
28. Han, S.; Mao, H.; Dally, W.J. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. *arXiv* **2015**, arXiv:1510.00149.
29. Liu, B.; Wang, M.; Foroosh, H.; Tappen, M.; Pensky, M. Sparse convolutional neural networks. In Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, Boston, MA, USA, 7–12 June 2015; pp. 806–814.
30. Wen, W.; Wu, C.; Wang, Y.; Chen, Y.; Li, H. Learning structured sparsity in deep neural networks. In Proceedings of the 30th International Conference on Neural Information Processing Systems, Red Hook, NY, USA, 5–10 December 2016; pp. 2082–2090.
31. Chang, Z.; Liu, S.; Xiong, X.; Cai, Z.; Tu, G. A survey of recent advances in edge-computing-powered artificial intelligence of things. *IEEE Internet Things J.* **2021**, *8*, 13849–13875. [CrossRef]
32. Weng, O. Neural Network Quantization for Efficient Inference: A Survey. *arXiv* **2021**, arXiv:2112.06126.
33. Bengio, Y.; Léonard, N.; Courville, A. Estimating or propagating gradients through stochastic neurons for conditional computation. *arXiv* **2013**, arXiv:1308.3432.
34. Ducasse, Q.; Cotret, P.; Lagadec, L.; Stewart, R. Benchmarking quantized neural networks on FPGAs with FINN. *arXiv* **2021**, arXiv:2102.01341.
35. 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 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, Monterey, CA, USA, 22–24 February 2017; pp. 65–74.
36. Fahim, F.; Hawks, B.; Herwig, C.; Hirschauer, J.; Jindariani, S.; Tran, N.; Carloni, L.P.; Di Guglielmo, G.; Harris, P.; Krupa, J.; et al. hls4ml: An open-source codesign workflow to empower scientific low-power machine learning devices. *arXiv* **2021**, arXiv:2103.05579.
37. Pappalardo, A.; Franco, G.; Nickfraser. Xilinx/Brevitas: CNV Test Reference Vectors r0. Available online: https://zenodo.org/records/3824904 (accessed on 16 April 2024).

38. Fan, H.; Ferianc, M.; Que, Z.; Li, H.; Liu, S.; Niu, X.; Luk, W. Algorithm and hardware co-design for reconfigurable cnn accelerator. In Proceedings of the 2022 27th Asia and South Pacific Design Automation Conference (ASP-DAC), Taipei, Taiwan, 17–20 January 2022; IEEE: Piscataway, NJ, USA, 2022; pp. 250–255.
39. Wang, Z.; Xu, K.; Wu, S.; Liu, L.; Liu, L.; Wang, D. Sparse-YOLO: Hardware/software co-design of an FPGA accelerator for YOLOv2. *IEEE Access* **2020**, *8*, 116569–116585. [CrossRef]
40. Haris, J.; Gibson, P.; Cano, J.; Agostini, N.B.; Kaeli, D. Secda: Efficient hardware/software co-design of fpga-based dnn accelerators for edge inference. In Proceedings of the 2021 IEEE 33rd International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD), Belo Horizonte, Brazil, 26–28 October 2021; IEEE: Piscataway, NJ, USA, 2021; pp. 33–43.
41. Li, F.; Zhang, B.; Liu, B. Ternary weight networks. *arXiv* **2016**, arXiv:1605.04711.
42. Zhou, A.; Yao, A.; Guo, Y.; Xu, L.; Chen, Y. Incremental network quantization: Towards lossless cnns with low-precision weights. *arXiv* **2017**, arXiv:1702.03044.
43. Holt, J.L.; Baker, T.E. Back propagation simulations using limited precision calculations. In Proceedings of the IJCNN-91-Seattle International Joint Conference on Neural Networks, Seattle, WA USA, 8–12 July 1991; IEEE: Piscataway, NJ, USA, 1991; Volume 2, pp. 121–126.
44. Courbariaux, M.; Bengio, Y.; David, J.P. Binaryconnect: Training deep neural networks with binary weights during propagations. In Proceedings of the 28th International Conference on Neural Information Processing Systems, Cambridge, MA, USA, 8–14 December 2015; pp. 3123–3131.
45. Park, J.; Sung, W. FPGA Based Implementation of Deep Neural Networks Using On-chip Memory Only. *arXiv* **2016**, arXiv:1602.01616. https://doi.org/10.48550/ARXIV.1602.01616.
46. Larkin, D.; Kinane, A.; O'Connor, N. Towards hardware acceleration of neuroevolution for multimedia processing applications on mobile devices. In Proceedings of the International Conference on Neural Information Processing, Hong Kong, China, 3–6 October 2006; Springer: Berlin/Heidelberg, Germany, 2006; pp. 1178–1188.
47. Farabet, C.; LeCun, Y.; Kavukcuoglu, K.; Culurciello, E.; Martini, B.; Akselrod, P.; Talay, S. Large-scale FPGA-based convolutional networks. *Scaling Mach. Learn. Parallel Distrib. Approaches* **2011**, *13*, 399–419.
48. Chen, Y.; Luo, T.; Liu, S.; Zhang, S.; He, L.; Wang, J.; Li, L.; Chen, T.; Xu, Z.; Sun, N.; et al. Dadiannao: A machine-learning supercomputer. In Proceedings of the 2014 47th Annual IEEE/ACM International Symposium on Microarchitecture, Cambridge, UK, 13–17 December 2014; IEEE: Piscataway, NJ, USA, 2014; pp. 609–622.
49. Courbariaux, M.; Bengio, Y.; David, J.P. Training deep neural networks with low precision multiplications. *arXiv* **2014**, arXiv:1412.7024.
50. Vanhoucke, V.; Senior, A.; Mao, M.Z. Improving the speed of neural networks on CPUs. In Proceedings of the Deep Learning and Unsupervised Feature Learning Workshop (NIPS 2011), Granada, Spain, 12–15 December 2011; pp. 1–8.
51. LeCun, Y.; Bottou, L.; Bengio, Y.; Haffner, P. Gradient-based learning applied to document recognition. *Proc. IEEE* **1998**, *86*, 2278–2324. [CrossRef]
52. Mohd, B.J.; Abed, S.; Hayajneh, T.; Alshayeji, M.H. Run-time monitoring and validation using reverse function (RMVRF) for hardware trojans detection. *IEEE Trans. Dependable Secur. Comput.* **2019**, *18*, 2689–2704. [CrossRef]
53. Zhang, X.; Heys, H.M.; Li, C. Fpga implementation and energy cost analysis of two light-weight involutional block ciphers targeted to wireless sensor networks. *Mob. Netw. Appl.* **2013**, *18*, 222–234. [CrossRef]
54. Mohd, B.J.; Hayajneh, T.; Yousef, K.M.A.; Khalaf, Z.A.; Bhuiyan, M.Z.A. Hardware design and modeling of lightweight block ciphers for secure communications. *Future Gener. Comput. Syst.* **2018**, *83*, 510–521. [CrossRef]
55. Alzubaidi, L.; Zhang, J.; Humaidi, A.J.; Al-Dujaili, A.; Duan, Y.; Al-Shamma, O.; Santamaría, J.; Fadhel, M.A.; Al-Amidie, M.; Farhan, L. Review of deep learning: Concepts, CNN architectures, challenges, applications, future directions. *J. Big Data* **2021**, *8*, 1–74.
56. Ghaffari, S.; Sharifian, S. FPGA-based convolutional neural network accelerator design using high level synthesize. In Proceedings of the 2016 2nd International Conference of Signal Processing and Intelligent Systems (ICSPIS), Tehran, Iran, 14–15 December 2016; IEEE: Piscataway, NJ, USA, 2016; pp. 1–6.
57. Rongshi, D.; Yongming, T. Accelerator implementation of lenet-5 convolution neural network based on fpga with hls. In Proceedings of the 2019 3rd International Conference on Circuits, System and Simulation (ICCSS), Nanjing, China, 13–15 June 2019; IEEE: Piscataway, NJ, USA, 2019; pp. 64–67.
58. Paul, D.; Singh, J.; Mathew, J. Hardware-software co-design approach for deep learning inference. In Proceedings of the 2019 7th International Conference on Smart Computing & Communications (ICSCC), Sarawak, Malaysia, 28–30 June 2019; IEEE: Piscataway, NJ, USA, 2019; pp. 1–5.
59. Deng, L. The mnist database of handwritten digit images for machine learning research [best of the web]. *IEEE Signal Process. Mag.* **2012**, *29*, 141–142. [CrossRef]
60. Intel. ModelSim-Intel FPGA Edition Software. Available online: https://www.intel.com/content/www/us/en/software-kit/750368/modelsim-intel-fpgas-standard-edition-software-version-18-1.html (accessed on 24 January 2024).
61. Intel. Stratix IV FPGAs Support. Available online: https://www.intel.com/content/www/us/en/support/programmable/support-resources/devices/stratix-iv-support.html (accessed on 20 January 2024).

62. Intel. Quartus Prime Software. Available online: https://www.intel.com/content/www/us/en/software/programmable/quartus-prime/download.html (accessed on 24 January 2024).
63. Mohd, B.J.; Hayajneh, T.; Khalaf, Z.A.; Vasilakos, A.V. A comparative study of steganography designs based on multiple FPGA platforms. *Int. J. Electron. Secur. Digit. Forensics* **2016**, *8*, 164–190. [CrossRef]
64. Mohd, B.J.; Hayajneh, T.; Khalaf, Z.A. Optimization and modeling of FPGA implementation of the Katan Cipher. In Proceedings of the Information and Communication Systems (ICICS), 2015 6th International Conference on, Amman, Jordan, 7–9 April 2015; IEEE: Piscataway, NJ, USA, 2015; pp. 68–72.
65. Ullah, S.; Rehman, S.; Shafique, M.; Kumar, A. High-performance accurate and approximate multipliers for FPGA-based hardware accelerators. *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.* **2021**, *41*, 211–224. [CrossRef]
66. Weste, N.H.; Harris, D. *CMOS VLSI Design: A Circuits and Systems Perspective*, 4th ed.; Pearson Education: London, UK, 2022.
67. Henriksson, M.; Gustafsson, O. Streaming Matrix Transposition on FPGAs Using Distributed Memories. In Proceedings of the 2023 IEEE Nordic Circuits and Systems Conference (NorCAS), Aalborg, Denmark, 31 October–1 November 2023; IEEE: Piscataway, NJ, USA, 2023; pp. 1–6.
68. Chu, T.; Luo, Q.; Yang, J.; Huang, X. Mixed-precision quantized neural networks with progressively decreasing bitwidth. *Pattern Recognit.* **2021**, *111*, 107647. [CrossRef]