Utilizing Cloud FPGAs towards the Open Neural Network Standard

Dimitrios Danopoulos[a,*,1], Christoforos Kachris[b,2] and Dimitrios Soudris[a,3]

[a]Department of Electrical and Computer Engineering, NTUA, Athens, Greece
[b]Democritus University of Thrace and ICCS-NTUA, Athens, Greece

ARTICLE INFO

Keywords:
Machine Learning
Neural Networks
ONNX
FPGAs
High Level Synthesis
Cloud
Heterogeneous Computing

ABSTRACT

Accurate and efficient Machine Learning algorithms are of vital importance to many problems, especially on classification or clustering tasks but need a universal AI model standard. Unifying machine learning models into a common ecosystem can lead to less development time and better framework interoperability. ONNX (Open Neural Network Exchange Format) is a popular open format to represent deep learning models so that AI developers can more easily move models between state-of-the-art tools. On top of that, hardware companies such as Nvidia or Intel try to keep up with this trend and produce hardware-optimized runtimes (i.e. for CPUs, GPUs, FPGAs) that can handle these open format AI models like ONNX. That enables developers to leverage an heterogeneous mix of hardware and use whichever AI framework they prefer. However, FPGAs have a more challenging solution strategy which as a platform it is also proven to address these kind of problems very efficiently in terms of performance and power. This work is based on an early development stage project which is called HLS4ML originally created for particle physics applications via the automatic generation of neural networks (NNs) for embedded Xilinx FPGAs. Our work involves a hardware-aware NN training and a generalized optimization scheme on top of HLS4ML that boosts the performance and power efficiency of this package and adds functionality for cloud FPGA firmware from any NN model. We start from the FPGA-oriented training of a model in Keras for image recognition, converting into the ONNX open format then porting and optimizing it for cloud FPGAs using a novel scheme with optimizations in host, memory and kernels while using multiple levels of network precision. To the best of our knowledge this is a novel approach that also achieves a speed-up of up to 102× over single CPU in performance and up to 5.5× over GPU in performance/watt.

1. Introduction

In recent years, Machine Learning based methods have achieved great success in a large number of applications and have been among the most powerful and widely used techniques in image/speech recognition, etc. Convolutional Neural Networks (CNNs) have gained significant traction due to their high accuracy and performance on visual recognition algorithms [19]. These are the most successful deep learning models that have revolutionized many industries infusing into increasingly more commercial products that have direct impact on people's everyday lives.

However, modern companies require a demanding workload of many terabytes of data or more to be processed every day due to these algorithms. The machine learning applications are constantly learning from real-world large scale data [7] and demand faster and faster processing speeds. This computational complexity motivated efforts to enhance these tasks using hardware-specific optimizations by leveraging different heterogeneous architectures combining platforms such as CPUs, GPUs and FPGAs [6, 22, 17]. The use of multicore systems [24] seems promising but the challenge of reducing the high energy cost and the processing times remains [5]. FPGA implementations on the other hand have seen great advancement as is it shown that they have been extremely effective on CNN tasks due to their massive parallelism and reconfigurability on the bit level. With the utilization of hardware accelerators the total throughput is increased due to the highly parallelizable massive number of multiply-accumulate operations (MACs) these algorithms need.

One of the major challenges in the hardware acceleration of CNNs is the deep learning framework interoperability. It is still hard for AI developers to move models between state-of-the-art tools and choose the combination that is best for them. Thus, there has been a move towards open format AI models such as ONNX that can present an ecosystem with common representations [8]. ONNX provides an open source format for AI models, both deep learning and traditional ML. It defines an extensible computation graph model, as well as definitions of built-in operators and standard data types [1]. It is also supported by many big companies like Alibaba, ARM, AWS, IBM, Huawei, Intel, Nvidia and many others. On top of this, porting such AI models for execution on platforms such as FPGAs can be very challenging and developers try to find ways to avoid rewriting code and overcome the code optimizations challenges. Hardware acceleration of neural networks on FPGAs with different network architecture each time is not a trivial task.

In this paper, we represent a novel scheme for automatic translation of CNNs for OpenCL FPGA devices in the Cloud. We add further support and optimizations on an early development open source project called hls4ml [9, 18] that consists of a compiler for high level language code on embedded Xilinx FPGAs out of neural network models. We present to the research community new architectures and optimization techniques for automatic translation of neural networks to cloud FPGAs all running fully in the FPGA hardware.

*Corresponding author
E-mail addresses
ORCID(s): 0000-0001-9327-5983 (D. Danopoulos)
[1]dimdano@microlab.ntua.gr (Dimitrios Danopoulos)
[2]kachris@microlab.ntua.gr (Christoforos Kachris)
[3]dsoudris@microlab.ntua.gr (Dimitrios Soudris)

Consequently, the development process of ONNX-based deep learning applications on FPGAs is faster and more power efficient, and facilitates neural network compilation and acceleration in general. In summary, the main contributions of the paper are as follows:

- We introduce a new optimization scheme for automatic generation of cloud FPGA firmware from open format ONNX models that is generalized for many neural network types with design space exploration capability.
- We introduce a new template of further optimizations on kernel, memory and host level for FPGAs such as the Xilinx Alveo U200 that was applied on top of *hls4ml* using a flexible heterogeneous streaming architecture with different precision between NN layers.
- We propose a hardware-specific training method of neural networks (a small MNIST-based and one bigger CIFAR-based were tested for demonstration purposes). We port the ONNX-converted models for automatic HLS translation for the Alveo board. We outperform other high performance devices (Xeon CPU, P100 GPU) in performance and performance/watt.

2. Related Work

CNN algorithms have been explored by many researchers especially for the software as well as the hardware optimizations that can be applied for a variety of applications like image recognition, object detection, etc. The following related work involves similar design methods for relatively similar problems or present a related problem that covers our problem domain.

Weijie You et al. [25] proposed a group-based DNN pipeline accelerator design for FPGAs. Specifically they evaluate the results in Alexnet and VGG16 networks with Xilinx ZC706 which is an embedded FPGA SoC in contrary with our device used in our paper which is a cloud FPGA. Also, C. Hao [14] proposed an FPGA/DNN co-design methodology with an automatic flow through an Auto-HLS engine to generate synthesizable C code of the FPGA, however they as well targeted an embedded FPGA, the Xilinx PYNQ-Z1. Moreoever, Sitao Huang [16] presents a design of a flexible sparse DNN inference accelerator on FPGA that can be configured for both mobile computing and high-performance computing scenarios, although no GPU comparison is done.

M. Ghasemzadeh et al. [11] presented a Residual Binarized Neural Network called ReBNet that runs on Xilinx FPGAs. This implementation uses 1-bit precision in each layer of the neural network doing the computations in a *Xnor-Popcount* style, however their algorithm lacks performance when compared with our solution. Even though their hardware accelerator is running on a bit different kernel frequency than ours, their CNN model is also based on the MNIST dataset. They state that they achieve a maximum of 64000 $Images/s$ while our MNIST model can achieve a maximum of 158000 $Images/s$ using 8-bit and higher precision weights while using half the resources.

Jiong Si et al. [23] tested a system for inference on FPGA platform using 8-bit and other precisions on MNIST dataset. While their FPGA runs on a lower frequency they still achieve a proportionally lower performance on 8-bit data than ours. They state that they used 25 Mhz clock while our FPGA attains the maximum available frequency of the device which is 300MHz. While our clock is 12× faster, it's essential to point that our design achieves a 60× speed-up compared with their proposed solution. Thus, one can state that our proposed system gives a 5× performance overall.

Alemdar et al. [2] implement fully-connected ternary-weight neural networks for FPGAs and report a latency of 20.5 μs on MNIST dataset. Using only ternary weights and activations, their ternary networks learn can inherently prune the smaller weights by setting them to zero during training. This makes them sparser and thus more energy-efficient. It's worth mentioning that the FPGA board used is a customized Xilinx Kintex 7, called Sakura-X which runs on 200MHz clock. Our board which uses similar resources and runs at 300MHz achieves a latency of 6.3 μs while using higher precision weights and activations.

Last, H.M. Makrani et al. [20] proposed a model to characterize diverse types of scale-out applications across a range of memory configuration parameters and maximize their performance/cost ratio in the cloud. Mena, the methodology proposed, navigates memory and processor parameters to find the system configurations for a given application and budget, although it does not reach the scope of FPGAs.

To conclude, a lot of work has been done using lower precision neural networks for FPGAs. However, on the next chapters we will describe a novel approach for FPGA deployment of CNNs which has the superiority of automatic generation of FPGA code from ONNX models with low latency and high power efficiency as well.

3. Background

Hardware-specific implementations of intense workloads (.i.e. on FPGAs) have shown great performance and performance per watt [3]. As we also show in the following sections, FPGAs can offer great reconfigurability especially in Neural Networks by enabling operations in lower fixed point arithmetic (compared with the 32-bit float of CPUs, GPUs) which eventually leads to less resources and more efficient designs [15]. The FPGA Performance Metric (FPM) is therefore increased as logic units decrease (LU) and designs permit higher frequencies ($FPM = \frac{f_{MAX}}{LU}$).

3.1. Neural Networks on Hardware

A lot of prior work on deploying neural networks to hardware exist both for FPGAs and CPUs, GPUs in order to speed-up the multiply-accumulate operations (MACs) found in the network computation. The benefit of the FPGAs however lies in their reconfigurable architecture which is capable of using pipelining for streaming processing through a CNN's layers. At the same time, an FPGA can take advantage of the lower precision custom arithmetic in its fabric which can reduce CNN parameters and speed-up computations.

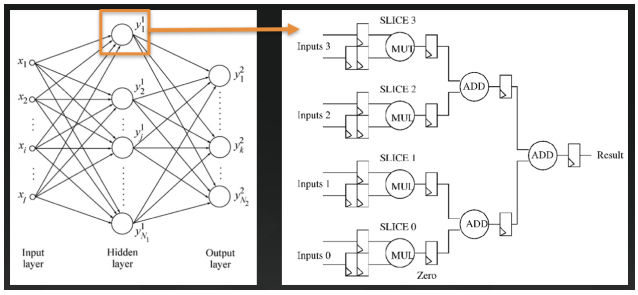

Figure 1: Hardware implementation of a neural network [21]

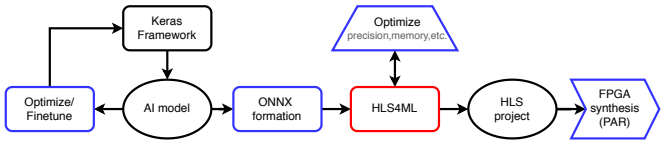

Figure 2: Design steps for ONNX model deployment to FPGAs. (Blue tasks indicate our work while red are taken from previous work. The HLS4ML python API was modified to support the new optimizations and changes.)

3.2. The ONNX format

Open Neural Network Exchange (ONNX) is an open ecosystem that empowers AI developers to ease the development of AI models. ONNX provides an open source format for AI models, both deep learning and traditional ML with an extensible computation graph model, as well as definitions of built-in operators and standard data types in order to be deployed for inference. ONNX is widely supported and can be found in many frameworks now such as Tensorflow, Caffe, Pytorch, MxNet, Matlab and it is supported by different hardware runtimes like Nvidia's, Qualcomm's, etc.

3.3. The "hls4ml" compiler package

Javier Duarte et al. [9] examined the use of FPGAs for fast inference in particle physics applications. They stated FPGA-based trigger and data acquisition systems have extremely low, sub-microsecond latency requirements that are unique to particle physics. So, they created a package to build machine learning models automatically for FPGAs using High Level Synthesis (HLS). The compiler automatically translates a trained neural network into HLS code based on the model architecture, weights and biases. It supports models trained in Keras, PyTorch or even ONNX. However, it's still in an early stage and has minimal support for layers. In our case, the model which we trained specifically for FPGA hardware could not compile correctly in hls4ml when we first tried it. It needed several modifications to avoid memory issues but it is still a good start of our solution strategy.

4. Design Challenges and Strategy

Machine learning models exported from frameworks like TensorFlow or PyTorch, require an efficient hardware with massive computation power thus a precise design flow is essential. For better understanding of the design strategy Figure 2 illustrates the proposed design flow. As we observe, everything starts with the training/optimization of an AI model on Keras using hardware-aware optimizations and then it is converted to ONNX format. Next, the open format NN model is generated into FPGA HLS code using HLS4ML package along with our NN and FPGA optimizations for cloud deployment. Last, the image recognition application runs on board with a tuned synthesis for high frequency and adequate PAR exploration.

4.1. Design Principles

Performance, modularity and re-configurability are critical for a high performance and versatile design.

4.1.1. Performance

In order to reduce the latency of the design and improve the efficiency of the whole application we used a streaming dataflow. Every layer is parallelized acting as a separate module and feeds its output to the next layer which accepts the previous output layout as input overlapping the previous layer's operation. Thus, we did not need to store each layer's intermediate results in off-chip memory since they are immediately passed down the stream which leads to a faster and more power efficient design. Also, the precision of the calculations was took into account in order to use the most efficient multipliers for the computations while keeping the accuracy error small.

4.1.2. Modularity

The design can accept different precision between the layers. Thus, we created an heterogeneous neural network with the appropriate accuracy in each layer depending on the activations needed. Also, each layer's precision on the weights, biases or activations can be changed seamlessly to fit the needs of the application accordingly. Last, as previously mentioned, every layer acts as a standalone accelerated module which can be controlled independently.

4.1.3. Re-configurability

HLS4ML must generate our custom FPGA firmware for a wide range of OpenCL FPGA devices with multiple copies of the same kernel for batch processing.

5. Architecture Design

We generally configured the FPGA according to FPGA design principles for high performance power efficiency along with a new custom OpenCL host API for accommodating cloud FPGAs for datacenters. Our semi-automated OpenCL-centric tool flow methodology for deploying neural networks on FPGA offers a range of distinct optimizations on software and hardware level as well. We first introduce a hardware-specific training of a neural network which generalizes to any neural network. This approach, as we will discuss next, offers great flexibility in the parallelization level on the DSPs especially when the neural networks weights are unsigned. Also, the translation of the generalized ONNX model format

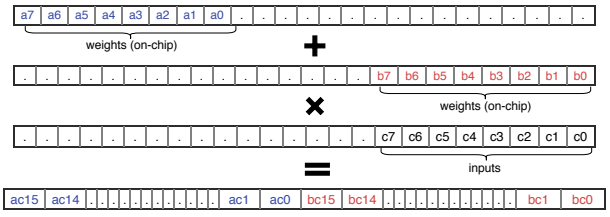

Figure 3: Packing two INT8 multiplications with a single DSP Slice

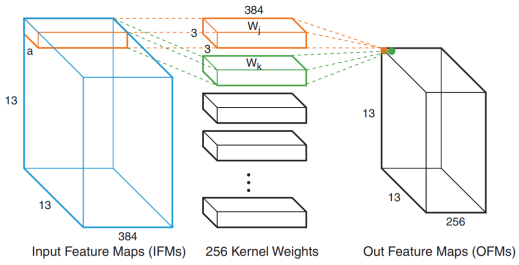

Figure 4: Illustration of input sharing technique [10]

to a fused hardware model with independent modules with each one acting as a layer gives the flexibility to parameterize and optimize further the neural network. Last, through the OpenCL host API we improved the memory accesses to the kernels using the maximum data bandwidth of the device. With the use of the OpenCL command queues and the precise synchronization between host and kernels high parallelization was gained at the coarse grain level.

5.1. Hardware Accelerator

Our hardware accelerator, as already mentioned consists of several accelerated modules as translated from the ONNX model which are pipelined (with II=1) and are connected serially in a streaming manner. Most important of which are the dense/convolution layers due to their computations (almost 90% of the total network execution time).

5.1.1. Multiplier optimization

We tried to keep 8-bit precision in all weights and biases (as usually they keep an acceptable inference accuracy [12, 13]) while the inner activations have mixed precision datatypes. A statistical analysis was done on the weight and activation parameters so as to determine the least required precision in each layer of the network.

A novel approach on the multiplier design was the method of inferring two multiplication operations efficiently within one clock cycle in one Xilinx's DSP48E2. The goal was to find a way to efficiently encode input a, b, and c so that the multiplication $(a + b) \times c$ can be easily separated into $a \times c$ and $b \times c$. We packed the two 8-bit inputs a and b in the 27-bit port of the DSP48E2 multiplier via the pre-adder so that the vectors are as far apart as possible as shown in Figure 3.

As we observe, the product of packed port and an 8-bit coefficient gives the result without the bits of each vector affecting the computation of one another. Our 8-bit input required a minimum of $16 + 8 = 24$-bit total input size so as to avoid any contamination of the upper bits by the computation of the lower bits.

5.1.2. Memory optimization

In Neural Networks the same set of inputs or weights are usually reused heavily in convolutional or dense layers. We chose to use *input sharing* as illustrated in Figure 4, thus form $a_i \times w_i$, and $a_i \times w_j$ type of parallel MAC operations. By shifting the values to the left using the discussed INT8 optimization technique, each DSP slice results in a partial and independent portion of the final output values.

Hence, the common c coefficient of the multiplication $a \times c + b \times c$ is the input a_i of the partial sum of products $O_k = \sum_{i=1}^{N} a_k w_{i,k}$ in a convolution layer. Moreover, we configured the neural network weights to be stored in the fast BRAMs of the FPGA device . We allocated the weights into matrix blocks in physically contiguous memory using the on-chip BRAMs, which are physically located near the computation of kernels. Thus, we guaranteed a fast communication allowing one-cycle read-writes of vectorized data with the most efficient data movers. Last, the blocks were fully partitioned (or partially if the network was big) so that the FPGA device can have simultaneous access to multiple weight values to enable massive parallelism.

5.1.3. Dataflow optimization

After optimizing the multipliers and memory, we proceeded with the final integration of the acceleration modules by running all the layers "one-shot" in a unified kernel with streaming architecture. We created channels based on FIFOs that allowed consumer layers to start operation before the producer layers have completed as illustrated in Figure 5. Each layer feeds its output to the next layer using similar datatype layouts and allows it to overlap in their operation once enough data has been accumulated in the previous layer, increasing the concurrency of the RTL design. Moreover, in streaming architecture there is no need to store each layer's intermediate results in off-chip memory since they are immediately passed down the stream. This resulted in a large resource reduction and power efficiency as we used the least resources needed for our design.

We also enabled batch processing for the forward operation so the user can insert a packed array of N images to process batched and output the result in a $N \times classes$ array which contains the classification probabilities. The *softmax* layer creates this array and outputs all the classification results after processing all images.

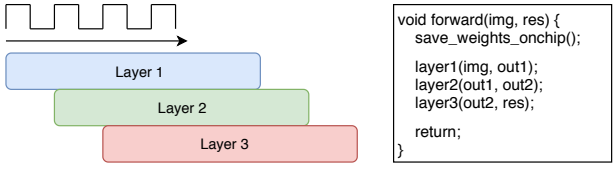

Figure 5: Task level pipelining between layers

5.2. Hardware-oriented NN training

As we already mentioned, we trained two custom neural networks for the purposes of demonstration, a small size MNIST-based network and a relatively bigger one on CIFAR dataset. The first one recognizes images of clothing which is a more challening task than the casual MNIST while the second recognizes CIFAR-10 images.

5.2.1. Model and parameters

First we scaled the image values to a range of 0 to 1 before feeding them to the neural network model by dividing by 255. We then split the dataset images in 60000 for the *training* and 10000 for the *test* set. The MNIST model consists mainly of 2 *dense* layers with more than $200K$ total synapses and *activation* layers in between while the other one is a VGG-16 variant for CIFAR-10 dataset which consists of 3 *convolution* and *pooling* layers with *activation* layers in between. We chose as an activation function for both models the *ReLU* function which is more stable, fast and efficient for FPGA hardware.

5.2.2. The sign extension issue

As previously discussed, we use an INT8 optimization on the multiplier of the hardware accelerator which uses a packed port to do 2 multiplications in one DSP. However, our neural network computations on the dense or convolution layers are originally done using signed by unsigned multiplications. This is due to all the positive activations from ReLU or input layer being positive numbers while the weights can be negative. When multiplying these type of numbers we need to sign-extend the inputs as seen below and this infers limitations. The sign extension is necessary to do the multiplications in unsigned format and then get the correct number in two's complement format otherwise the result can be invalid.

For example, when we multiply two 8-bit numbers with one of them being signed, the sign extension leads to a 16 by 8 bit multiplication in order for the result to be in the correct two's complement form. This applies for integers but also fixed point numbers as seen in Figure 7 because we treat them as integers and later we just determine their position of the binary point. In this regard, we cannot by default apply efficiently our INT8 optimization because the packed port for the single DSP multiplication cannot fit two 16-bit numbers.

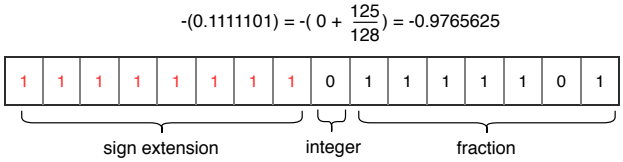

Figure 7: Sign extension for an 8-bit fixed point number

5.2.3. The weight regulation

One way to overcome the previous problem is to force the Tensorflow/Keras framework which we train our models to put specific constraints during training. It is a type of $L2$ regularization which helps control the weight growth of the model. We applied a *non-negativity* constraint on a per-layer basis both on weights and biases. This worked poorly at first as some weights became large positive numbers after training as the result from overfitting but adding further guides (weight/bias initialization) or more epochs was essential.

Last, a couple of retraining steps and pruning were needed which led to an $[0, 2.5]$ weight distribution with most weights found in the $[0, 0.05]$ space. At the end, at post-training, we trimmed the integer width of the 8-bit fixed point values keeping only 1 bit of integer information for the weights as the values that needed 2 integer bits were very few.

5.3. Final System Design using OpenCL

After achieving a low latency design using fine grain parallelism on the kernel level within the PL fabric, we designed an efficient OpenCL host API using standard OpenCL API calls. More specifically, we allocated the image inputs in C++ vectors with size $28 * 28 * N$ where N is the number of images if the user prefers batch processing. In this way, the image matrix was allocated in a physically contiguous memory and allowed us to use the most efficient data movers for the data transfers to and from DDRs. In order to maximize data throughput we implemented a 512-bit user interface on each kernel side as well which is the maximum memory bandwidth supported in Xilinx OpenCL FPGAs. At the same time we used all DDRs of the device simultaneously reaching maximum data transfer speeds. Also, concurrency from OpenCL command queues that initiate the kernels and synchronization between host and kernels were all precisely constructed to run efficiently in a constant dataflow.

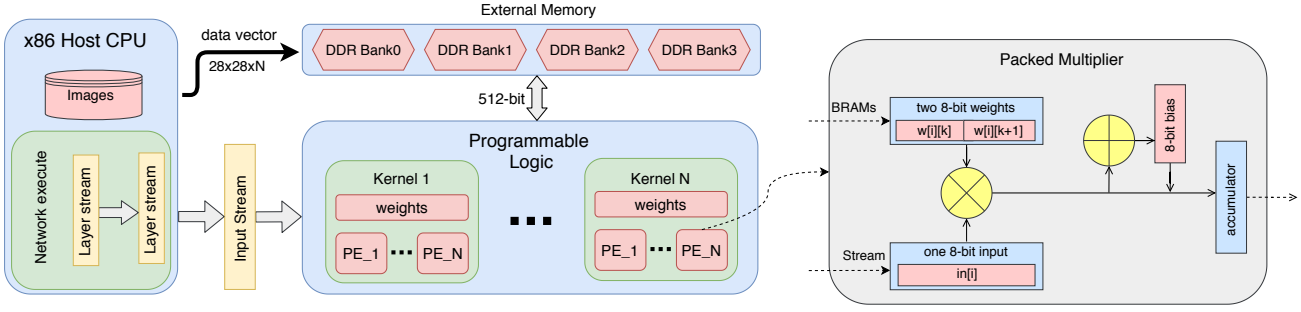

Figure 6: Final System dataflow

Last, we tried to avoid SLR crossing wherever possible as this leads to less efficient designs in terms of latency and power due to longer critical paths created but usually our kernels were not big enough to exceed each SLR resources. In Figure 6 we can observe the whole system from the host CPU then the FPGA, up to the calculations done in the multiplier level in each Processing Element (PE). It's worth mentioning that the final system with all optimizations described was integrated into HLS4ML package (in the python API) so that it implements our custom architecture with the same automatic fashion as previously with no extra limits.

6. Evaluation and Results

In this section we will evaluate and profile our application. We will start from the illustration of the hardware-aware training of a neural network which we used as a test case and evaluate its accuracy. Note that post-training quantization was also applied before calculating the final inference accuracy. Last we will proceed with the performance evaluation of the hardware accelerator and the final working system on the small MNIST model and the bigger CIFAR-10 model.

6.1. Neural Network Model

For the purposes of demonstration, we implemented two custom neural networks as described previously using Keras [4] library. These models were selected because they can be often found in the cloud industry (clothing classification and object recognition respectively). On Figure 8 we can observe the validation accuracy of the first one acquired for every training step between the default and the regulated model.

The lower accuracy achieved in the hardware-optimized model is due to the weight initialization and constraints we put in order to fully take advantage of our proposed multiplier unit and eventually do double packed MAC operations. The validation accuracy difference is $\sim 4\%$ between the two models while the inference accuracy on the final system (using mainly 8-bit weights and activations) has only a 6% drop from the original model on average, which is a decent price to pay for the double performance acquired with the same resources along with the 8-bit precision benefits (fewer resources, lower latency and power).

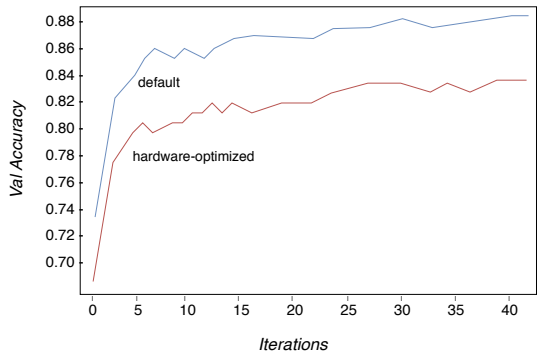

Figure 8: Accuracy comparison between neural network models

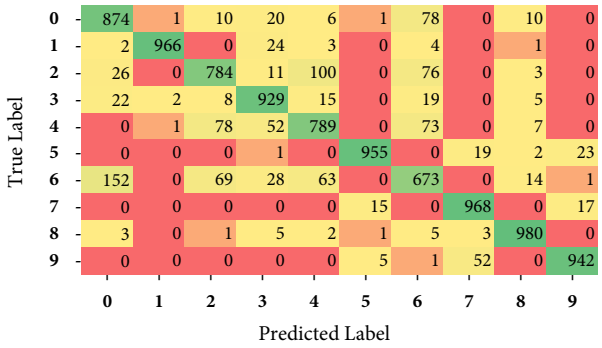

Figure 9: Heatmap on the confusion matrix of the classification model

Also, in our particular case, we chose to visualize the classification accuracy in a confusion matrix (see Figure 9), because there is the problem of the typical accuracy ratio of hiding the detail of the classification model. A confusion matrix is the number of correct and incorrect predictions summarized with count values and broken down by each class. This gave us an insight not only into the errors being made by our classifier but also by our hardware optimizations that we did.

6.2. Accelerator Performance

For the evaluation of the design we first confirmed the correctness of the accelerator, specifically our custom multiplier because wrong DSP output could ruin the functionality of the whole CNN. The hardware setup for the evaluation was a Xilinx Alveo U200 FPGA. The system has 64GB off-chip RAM of 77 GB/s bandwidth and was installed on Gen3x16 PCI express running at a kernel clock of 300MHz. Table 1 shows the resource utilization and timing for each NN layer (per layer type) in the case of the MNIST model.

Table 1
FPGA resource utilization and latency per layer

	Utilization summary				Timing	
Layer	***BRAM***	***DSP***	***FF***	***LUT***	***Latency***	***FPS***
Dense	192	64	2449	12656	1709	-
ReLu	0	0	16	127	130	-
Softmax	7	0	985	2401	51	-
Total	199	64	3450	15184	1890	158K

Our streaming architecture has a total latency almost equal to the sum of the initiation intervals of each layer which equals to 1890 kernel cycles. The kernel also reached the maximum device frequency of 300 MHz but theoretically can reach up to 400 MHz as seen from the Worst Negative Slack (WNS) which was $0.8ns$. So, our neural network accelerator can do a full forward pass in 1890 kernel cycles or $6.3\mu s$ which translates to 158000 FPS. For instance, consider a full HD 1920×1080 video stream at 30 FPS, which is to be chopped up into 28×28 tiles for neural network inference. Handling this task with real-time performance would require a NN inference rate of 80000 FPS, which is successfully handled by our system.

Table 2
Evaluation vs other Architectures

Device information			Performance Evaluation				Power Evaluation	
System	Model	Architecture	CIFAR	Speed-up	MNIST	Speed-up	Watt(avg)	Perf./Watt (max)
CPU	Xeon 2.4GHz	22-nm	58 ms	1x	43 ms	1x	9 W *	1x
GPU	Nvidia P100	16-nm	1.1 ms	52.7x	0.75 ms	57x	95 W	5.4x
FPGA	Alveo U200	16-nm	2.7 ms	21.5x	0.42 ms	102.3x	31 W	29.7x

* Scaled to single-core

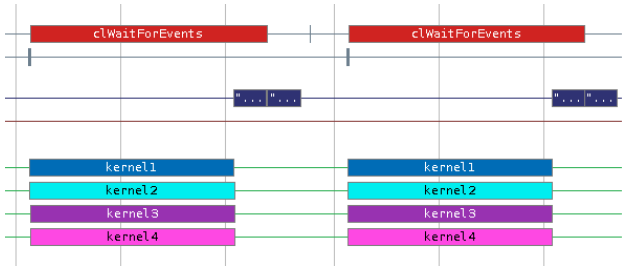

Figure 10: Task level parallelism with concurrent kernel execution

Last, in Figure 10 we can see the coarse-grained concurrency on the task level parallelism using 4 kernels in parallel with the data being computed and transferred simultaneously. The illustration was exported directly from SDAccel framework.

6.3. Final System Performance

For the final system evaluation we first tested the *end-to-end* execution time for a single forward pass of our two neural networks (MNIST and CIFAR) including the memory transfers. We then compared the results with the execution time on a single-core Xeon CPU and an Nvidia Tesla P100 GPU (the two neural network timings might appear similar due to device overhead). The final FPGA system as seen from Table 2 is superior in terms of performance and average power consumed especially for the smaller MNIST model (more discussion in the next paragraph). Our system uses reduced-precision weights and activations instead of the default 32-bit floating point, and combined with the use of the packed multiplier we have a system with minimal latency which is essential for critical applications. Also, it is worth mentioning that our design uses a very small percentage of the device resources which can be very power efficient, hence can be also deployed to smaller embedded FPGA SoCs besides the datacenters. This is essential for critical applications that require low latency and power.

6.3.1. Results Evaluation

In order to have a comparison of our systems' performance with other work on the same architecture we carefully compared other projects which involved FPGA designs for similar problems or presented a related problem that covered our problem domain (see more in Related Work). The most similar was an 8-bit accelerator on MNIST from Jiong Si et al. [23] which our design surpassed with 5× in performance.

In the case of comparing with different architectures we performed the same benchmarks on a CPU and GPU system with the results shown in Table 2. The baseline used in every metric was the Xeon CPU. We can observe that while GPU outperforms the other two systems on the bigger CIFAR10 model, our FPGA design has the best performance in the MNIST model. It manages a 102.3× speed-up from CPU and 1.8× from GPU. Moreover, in terms of power efficiency, the superiority of our FPGA architecture is shown in the Performance/Watt metric where it manages to have a 29.7× speed-up from CPU and 5.5× from GPU. It's worth noticing that the power efficiency of the FPGA is higher for both model cases (CIFAR and MNIST).

7. Conclusion

In this work we proposed a new re-designed framework that can automatically generate FPGA firmware in High Level Synthesis from neural network models. We implemented several optimizations and added further functionality in the preexisting *hls4ml* package making it faster and more efficient. Also, for demonstration purposes, we trained and fine-tuned two custom neural networks (one small and one larger in size) with hardware-oriented optimizations which could classify images of clothing or objects, typical applications that can be found in the cloud industry. Our work also showed that the proposed architecture can surpass the performance and power efficiency of other high end platforms like CPU or GPU. Moreover, the accelerator performance was also compared with other FPGA designs in Related Work (reduced precision NNs, etc.) and showed superiority. From a research point of view, we are focusing on boosting more the performance and adding further features in the package like an integrated hw-aware training as described in this work. The spectrum of possible design space tradeoffs is vast but this work shed some light to the area with successful results aiming to make FPGAs contribute fundamentally into this open software-hardware ecosystem.

Acknowledgment

This project was funded from the Xilinx University program and the Hellenic Foundation for Research and Innovation (HFRI) and the General Secretariat for Research and Technology (GSRT), under grant agreement No 2212- Hardware Acceleration of Machine Learning Applications in the Cloud.

References

[1] Abdelouahab, K., Pelcat, M., Sérot, J., Berry, F., 2018. Accelerating CNN inference on fpgas: A survey. CoRR abs/1806.01683. URL: http://arxiv.org/abs/1806.01683, arXiv:1806.01683.

[2] Alemdar, H., Leroy, V., Prost-Boucle, A., Pétrot, F., 2017. Ternary neural networks for resource-efficient ai applications, pp. 2547–2554. doi:10.1109/IJCNN.2017.7966166.

[3] Betkaoui, B., Thomas, D.B., Luk, W., 2010. Comparing performance and energy efficiency of fpgas and gpus for high productivity computing, in: 2010 International Conference on Field-Programmable Technology, pp. 94–101.

[4] Chollet, F., et al., 2015. Keras. URL: https://keras.io.

[5] Cong, J., Fang, Z., Lo, M., Wang, H., Xu, J., Zhang, S., 2018. Understanding performance differences of fpgas and gpus, pp. 93–96. doi:10.1109/FCCM.2018.00023.

[6] Danopoulos, D., Kachris, C., Soudris, D., 2018. Acceleration of image classification with caffe framework using fpga, in: 2018 7th International Conference on Modern Circuits and Systems Technologies (MOCAST), pp. 1–4. doi:10.1109/MOCAST.2018.8376580.

[7] Danopoulos, D., Kachris, C., Soudris, D., 2019. Approximate Similarity Search with FAISS Framework Using FPGAs on the Cloud. pp. 373–386. doi:10.1007/978-3-030-27562-4_27.

[8] Danopoulos, D., Kachris, C., Soudris, D., 2020. Automatic generation of fpga kernels from open format cnn models, in: 2020 IEEE 28th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM), pp. 237–237.

[9] Duarte, J., Han, S., Harris, P., Jindariani, S., Kreinar, E., Kreis, B., Ngadiuba, J., Pierini, M., Rivera, R., Tran, N., Wu, Z., 2018. Fast inference of deep neural networks in FPGAs for particle physics. Journal of Instrumentation 13, P07027–P07027. URL: https://doi.org/10.1088%2F1748-0221%2F13%2F07%2Fp07027, doi:10.1088/1748-0221/13/07/p07027.

[10] Fu, Y., Wu, E., Sirasao, A., Attia, S., Khan, K., Wittig, R., 2017. White paper: Deep Learning with INT8 Optimization on Xilinx Devices. Technical Report WP486 (v1.0.1). Xilinx. URL: https://www.xilinx.com/support/documentation/white_papers/wp486-deep-learning-int8.pdf.

[11] Ghasemzadeh, M., Samragh, M., Koushanfar, F., 2018. Rebnet: Residual binarized neural network, in: 2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM), pp. 57–64. doi:10.1109/FCCM.2018.00018.

[12] Gysel, P., Motamedi, M., Ghiasi, S., 2016. Hardware-oriented approximation of convolutional neural networks. CoRR abs/1604.03168. URL: http://arxiv.org/abs/1604.03168, arXiv:1604.03168.

[13] Han, S., Mao, H., Dally, W., 2016. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding.

[14] Hao, C., Zhang, X., Li, Y., Huang, S., Xiong, J., Rupnow, K., Hwu, W.m., Chen, D., 2019. Fpga/dnn co-design: An efficient design methodology for iot intelligence on the edge, pp. 1–6. doi:10.1145/3316781.3317829.

[15] Hettiarachchi, D.L.N., Davuluru, V.S.P., Balster, E.J., 2020. Integer vs. floating-point processing on modern fpga technology, in: 2020 10th Annual Computing and Communication Workshop and Conference (CCWC), pp. 0606–0612.

[16] Huang, S., Pearson, C., Nagi, R., Xiong, J., Chen, D., Hwu, W., 2019. Accelerating sparse deep neural networks on fpgas, in: 2019 IEEE High Performance Extreme Computing Conference (HPEC), pp. 1–7.

[17] Kachris, C., Soudris, D., 2016. A survey on reconfigurable accelerators for cloud computing, in: 2016 26th International Conference on Field Programmable Logic and Applications (FPL), pp. 1–10. doi:10.1109/FPL.2016.7577381.

[18] Kreis, B., Tran, N., Duarte, J., 2019. Machine learning in fpgas using hls.

[19] Krizhevsky, A., Sutskever, I., Hinton, G., 2012. Imagenet classification with deep convolutional neural networks. Neural Information Processing Systems 25. doi:10.1145/3065386.

[20] Makrani, H.M., Homayoun, H., 2017. Mena: A memory navigator for modern hardware in a scale-out environment, in: 2017 IEEE International Symposium on Workload Characterization (IISWC), pp. 2–11.

[21] Omondi, A.R., Rajapakse, J.C., 2006. FPGA Implementations of Neural Networks. Springer-Verlag, Berlin, Heidelberg.

[22] Putnam, A., Caulfield, A.M., Chung, E.S., Chiou, D., Constantinides, K., Demme, J., Esmaeilzadeh, H., Fowers, J., Gopal, G.P., Gray, J., Haselman, M., Hauck, S., Heil, S., Hormati, A., Kim, J., Lanka, S., Larus, J., Peterson, E., Pope, S., Smith, A., Thong, J., Xiao, P.Y., Burger, D., 2015. A reconfigurable fabric for accelerating large-scale datacenter services. IEEE Micro 35, 10–22. doi:10.1109/MM.2015.42.

[23] Si, J., Harris, S.L., 2018. Handwritten digit recognition system on an fpga, in: 2018 IEEE 8th Annual Computing and Communication Workshop and Conference (CCWC), pp. 402–407. doi:10.1109/CCWC.2018.8301757.

[24] Sze, V., Chen, Y.H., Einer, J., Suleiman, A., Zhang, Z., 2017. Hardware for machine learning: Challenges and opportunities, pp. 1–8. doi:10.1109/CICC.2017.7993626.

[25] You, W., Chen, D., Wu, C., 2019. A flexible dnn accelerator design with layer pipeline for fpgas, in: 2019 6th International Conference on Information Science and Control Engineering (ICISCE), pp. 959–962.

Dimitrios Danopoulos is a PhD canditate at the School of Electrical and Computer Engineering of the National Technical University of Athens (NTUA). His research area involves the development and hardware acceleration of Machine Learning applications in the cloud for data centers. Also, he has received his electrical and computer engineering diploma degree at NTUA. During his studies, he majored in sciences that mainly concerned software and computing systems, but bioengineering as well.

Christoforos Kachris is a senior research associate at ICCS/NTUA. He is the editor of the book Hardware Accelerators in Data Centers. He has over 15 years of experience on FPGAs (reconfigurable computing), digital design, embedded systems (SoCs), and HW/ SW co-design mainly in network processing, microservers, optical interconnects, and telecommunication systems. He also has more than 10 years experience in EU-funded projects (proposal preparation/writing, researcher, principal investigator, WP leader. He was the technical project manager of VINEYARD h2020 project on the efficient utilization of FPGAs.

Dimitrios Soudris received his diploma and PhD in Electrical Engineering from the University of Patras in 1987 and 1992, respectively. He is currently working as a Full Professor in the School of Electrical and Computer Engineering, National Technical University of Athens, Greece. He is project coordinator and/or and principal investigator in numerous research & development projects (>55) funded by the European Commission, ENIAC-JU, European Space Agency, the Greek Goverment and the European and Greek Industry. His research interests include embedded systems design, reconfigurable architectures, reliability and low power VLSI design.