Precision Matters: FPGA-Driven Hardware Acceleration of Quantized Deep Learning Models

Mobin Vaziri*, Seyed M Mohtasebi†
* Department of Computer and Software Engineering, Polytechnique Montréal
† Department of Mathematics and Industrial Engineering , Polytechnique Montréal
{mobin.vaziri, seyed-mojtaba.mohtasebi}@polymtl.ca

***Abstract*—This study presents a novel approach to optimize the implementation of Deep Neural Networks (DNNs) on Field-Programmable Gate Arrays (FPGAs) using High-Level Synthesis (HLS) and model compression techniques. The proposed workflow, which includes hardware-aware pruning and quantization, has significantly improved both the model's accuracy and the hardware cost. The optimized model achieved a minimal accuracy drop of 0.84% for 75% sparsity and INT8 quantization, while reducing the LUT utilization, FF utilization, and DSP utilization by 39%, 13%, and 100% respectively. The study has made significant strides in addressing the challenges of implementing DNNs on FPGAs, offering a promising solution for accelerating the hardware design process, optimizing the hardware implementation, and validating the effectiveness and versatility of the developed model. The source code of this work is available on GitHub[1].**

I. Introduction

Deep Neural Networks (DNNs) are becoming increasingly popular for solving complex tasks in various fields. However, as the tasks' difficulty increases, the models must become deeper and more computationally extensive to achieve accurate results. This problem results in the impracticality of these models for real-time implementation on edge devices such as Field-Programmable Gate Arrays (FPGAs). FPGAs are renowned for their lower energy consumption and higher performance than CPUs and GPUs. Nevertheless, their limited computational resources pose a significant challenge, particularly when meeting strict throughput and latency requirements. Another issue is their development time, which designing and debugging in Register-Transfer Level (RTL) needs expertise and for larger and more complex models this process takes an unexpectedly long time. This project aims to address these problems with rapid hardware design using High-level Synthesis (HLS) and model compression techniques to implement the DNNs on FPGAs efficiently. The main contributions of this study are:

- Accelerate the hardware design process of DNNs through HLS modeling.
- Optimize the hardware implementation through hardware-aware quantization and pruning.
- Validate the effectiveness and versatility of the developed model by conducting extensive experiments on a dedicated hardware accelerator.

[1]https://github.com/smartward3n/ELE6310E-Efficient-HW-Implem.-of-Deep-Neural-Networks

The paper is structured in the following manner. In Section II, we discuss the related works and our motivation for conducting this study. The workflow is presented in detail in Section III, and the results of our experiments are provided in Section IV. Finally, we conclude the paper in Section V and present our future works.

II. Related Works

Existing frameworks such as VitisAI [1] and FINN [2] have emerged to enable the deployment of quantized deep neural network models on FPGAs. Quantization is a key technique employed by these frameworks, which involves storing weights and activation tensors in lower bit precision than standard training precision, leading to memory and computational cost savings. There are two ways to apply quantization to a model, Quantization Aware Training (QAT) and Post-Training Quantization (PTQ). QAT integrates quantization techniques during model training to maintain original accuracy, while Post-Training Quantization applies quantization after training, trading off accuracy for reduced model size without additional training time. However, while these frameworks are optimized for AMD-Xilinx cores, they lack support for QAT and are limited to specific data types such as INT8 datatype. This limitation may result in obtaining lower accuracies and underutilization of FPGA's reconfigurability for arbitrary data types.

III. Methodology

Our proposed workflow, illustrated in Fig. 1, commences with creating and training a DNN model using a single-precision datatype to obtain the baseline accuracy. Subsequently, we embark on compressing the model employing a pruning technique, aiming to pinpoint the optimal balance between compression ratio and accuracy degradation by exploring various sparsity levels. After pruning, the next step involves QAT applied to the pruned model, utilizing different datatypes. The objective is to attain the optimal model configuration with the lowest bitwidth, which minimizes accuracy reduction. Once the optimal model is achieved, we employ an HLS converter to transform the high-level, software-defined model into an HLS synthesizable model. With the generated HLS model, we can then compile and generate the RTL model of our DNN, ready for implementation on the target hardware, which, in this study, is an FPGA. In the subsequent

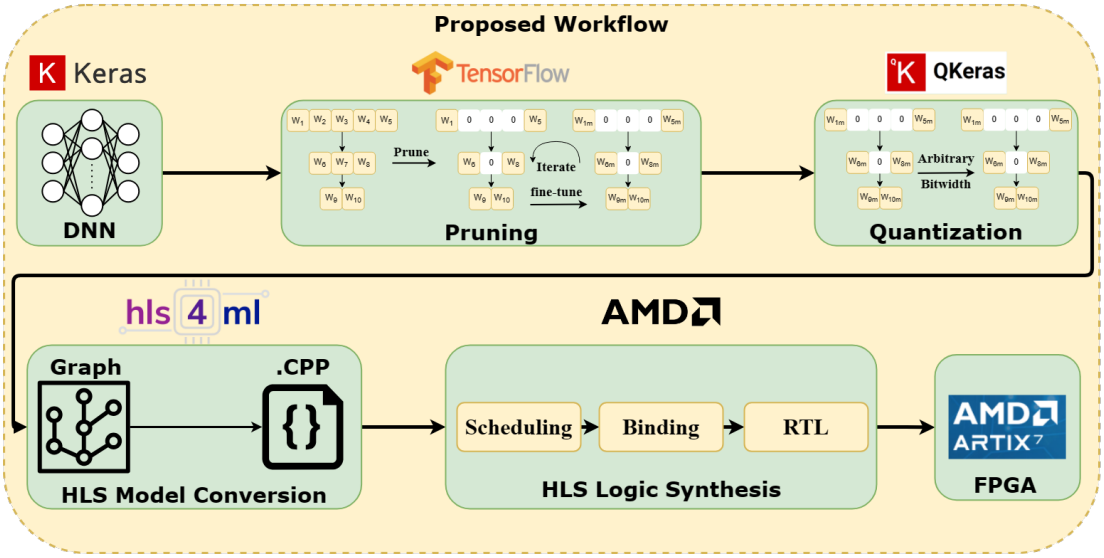


Fig. 1: The outline of the project workflow.

TABLE I: The Hyperparameters of the DNN

Layer (type)	Output Shape	#Parameter	Activation
Input	16	-	-
fc1 (Dense)	64	1088	ReLU
fc2 (Dense)	32	2080	ReLU
fc3 (Dense)	32	1056	ReLU
Output (Dense)	5	165	Softmax

subsections, we will delve into the intricacies of each workflow block.

A. DNN Model Development

In particle physics experiments such as those at the Large Hadron Collider (LHC), vast amounts of complex data from collision events pose challenges for traditional analysis methods [3]. DNNs offer a solution by automatically extracting intricate patterns and relationships from this data. With interconnected layers of neurons, DNNs can model complex relationships, making them promising for analyzing particle physics data. However, in the hardware triggering system in a particle detector at the CERN LHC, we have strict requirements for latency and power. The DNN model should make decisions in 10 μs or even lower than this threshold.

In this study, We use the "hls4ml_lhc_jets_hlf" dataset [3] containing 830,000 samples, each with 16 features capturing moment-based variables of jets observed in high-energy physics experiments. The target is the categorical classification of each jet into one of five classes. We split the data into 80% training, and 20% testing sets to train and evaluate the model's performance. In Table I the model with its parameters is presented in detail.

B. Structured Pruning

Structured pruning is a highly effective technique used to reduce the size of neural networks by identifying and removing entire neurons or connections based on specific criteria such as low importance [4]. In contrast to unstructured pruning, which removes individual weights without considering their spatial or structural relationships, structured pruning targets specific patterns or groups within the network. This optimization is particularly beneficial for hardware implementations, as it enables efficient memory access and streamlined processing, utilizing the regularity of structured patterns for accelerated computations. TensorFlow's constant sparsity feature is an excellent tool for implementing structured pruning [5]. It maintains a fixed percentage of zero-valued weights within the network while the model is in the training phase. This feature is particularly useful when compiling the model to an RTL model using HLS, as it enables the compiler to detect zero weights and avoid allocating computational resources to multipliers. The expected outcome of this feature is a significant reduction in computation resources in FPGAs, which will be elaborated upon in detail in Section IV.

C. Uniform Quantization & QAT

Uniform quantization involves the process of reducing the precision of weights and activations to a finite set of discrete levels. This technique is commonly employed to minimize memory requirements and computational complexity while maintaining acceptable model performance. By quantizing parameters uniformly, typically to a power-of-two scale, DNNs can be deployed more efficiently on hardware with limited resources such as mobile devices or embedded systems. Additionally, it facilitates the implementation of fixed-point arithmetic operations, which further accelerates inference speed. As previously noted, the problem with previous frameworks is they are limited to specific datatypes and do PTQ which

without considering the dynamic range of the model's parameters results in inaccuracies. In this work, we integrate QKeras into our workflow [6]. QKeras is a Python library developed by Google that extends TensorFlow and Keras functionalities to support quantization-aware training and inference for neural networks. By leveraging QKeras, we can uniformly quantize our DNN model with arbitrary bit widths. This flexibility makes QKeras an excellent choice for custom hardware accelerators such as FPGAs and Application-Specific Integrated Circuits (ASICs), as their resources can be tailored based on the defined bit width.

D. HLS & hls4ml framework

In our study, we utilized hls4ml, a framework that can translate high-level models from popular frameworks such as Keras, PyTorch, and ONNX into HLS models [3]. It helps in estimating hardware costs and timings across various stages, including unoptimized, pruned, quantized, or a combination of these. Before we dive deep into hls4ml, it is essential to understand the basics of HLS. HLS involves expressing the algorithmic behavior of a model using a high-level language such as C++. Then, this description is translated into RTL models in two phases - scheduling and binding. Scheduling determines when each task in the algorithm should execute, whether sequentially or in parallel, and allocates particular timing for task execution. Binding assigns resources to these tasks, deciding which hardware components like registers and computational units are utilized during execution, optimizing resource utilization. HLS compilers provide control over hardware design through directives called pragmas. They enable optimizations, such as loop unrolling and pipelining. One such optimization in hls4ml is increasing the reuse factor, which reduces area utilization overhead by reusing computational resources like multipliers and adders. For instance, in fully connected neural networks (FCNNs), loop execution can be adjusted using **#pragma** HLS unroll, where the unrolling factor determines the level of concurrency and resource reuse [see Algorithm 1]. Table II, presents the results for implementing our DNN model with different reusing factors. As it can be seen, without reusing components (Reuse factor = 1) the model can not be fitted as the number of DSPs exceeds the device capacity, and increasing this factor to 8 can introduce huge latencies due to stalls from reusing components. While this classical optimization plays a crucial role in reducing area overhead, it can impact latency and throughput, particularly in applications that require rapid decision-making. In Section IV, we elaborate on how model compression techniques can mitigate hardware overhead while maintaining a fully pipelined design without the need for reusing computational resources.

The high-level architecture considered by hls4ml for its accelerators involves keeping all model parameters inside the FPGA and utilizing the AXI4-Stream interface for reading input features and writing back the results. In hls4ml, the model parameters are defined as C arrays, and we have different options to map them, such as on-chip memories (Block RAMs), registers (Flip-Flops), or Look-Up Tables. We can control this mapping using the **#pragma** HLS bind_storage and directly instruct the compiler where to hold the parameters. However, when we apply pragmas such as loop unrolling and pipelining, HLS behaves as a black box and may map the parameters to other resources, disregarding our pragma instructions. HLS designers cannot delve into the architecture description due to the underlying HLS optimizations applied to meet user criteria.

Algorithm 1 One layer HLS model for FCNNs

Input: $inp\_ns,\ weights,\ bias$
Output: out_ns

procedure FCNN (inp_ns, weights, bias)
  $acc_1 \leftarrow 0$ // Define an accumulator
  for $i \leftarrow 0$ to out_ns **do**
    #pragma HLS Unroll factor=out_ns
    $acc_1 \leftarrow 0$
    for $j \leftarrow 0$ to inp_ns **do**
      #pragma HLS Unroll factor=inp_ns
      $acc_1 \leftarrow \text{MAC}(inp\_ns[j], weights[j][i])$
    end for
    $acc_1 \leftarrow acc_1 + bias[i]$
    $out\_ns[i] \leftarrow acc_1$
  end for
end procedure

E. FPGA Implementation

FPGAs are a type of integrated circuit that can be reconfigured after manufacturing to suit specific applications. This flexibility makes them useful for a wide range of tasks, from signal processing to machine learning. Configurable logic blocks (CLBs) are an important part of FPGA architecture, allowing the user to implement any logical functionality within the chip. Logic elements, such as Look-Up Tables (LUTs) and Flip-Flops (FFs), play crucial roles in facilitating the execution of complex logical functions within cLBs. LUTs are programmable memory blocks that store predefined truth tables, allowing for rapid computation of logic functions. Meanwhile, FFs serve as sequential logic elements, enabling the storage and manipulation of binary data. FPGAs also feature Block RAMs (BRAMs) for on-chip data storage, improving computational efficiency and reducing latency. Digital Signal Processors (DSPs) are specialized units that are optimized for arithmetic operations and are ideal for signal processing tasks. The AMD-Xilinx 7th generation of FPGAs includes the Artix,

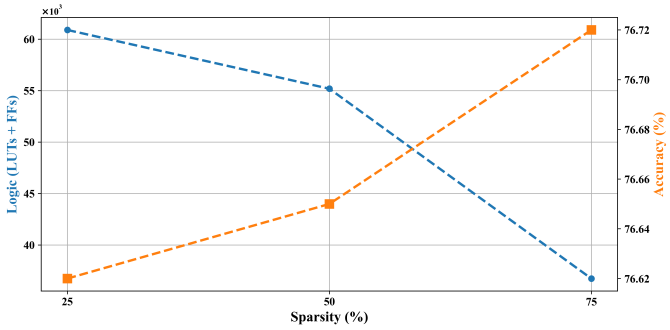
TABLE II: Adjusting reuse factor

	Hardware Utilization					
Reuse Factor	**LUT (%)**	**FF (%)**	**DSP (%)**	**BRAM (%)**	**Latency† (Clock Cycles)**	**Interval† (Clock Cycles)**
1	50	19	148	1	41	1
4	45	16	126	1	45	4
8	39	18	71	1	52	8

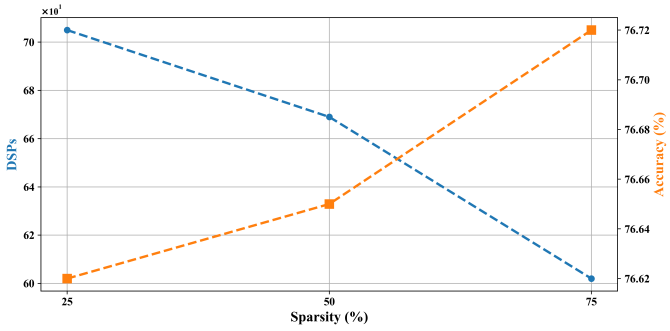
† Clock cycle is set to 5ns.

TABLE III: Hardware Cost and Timings for Pruned + Quantized Models

Datatype	Hardware Utilization						Model Performance	
	LUT (%)	FF (%)	DSP (%)	BRAM (%)	Latency (CC)	interval (CC)	Accuracy (%)	AUC (%)
INT16	12	6	91	1	35	1	76.23	94.08
INT8	11	6	0	1	33	1	75.66	93.76
INT6	9	4	0	1	32	1	75.13	93.37
INT4	7	3	0	1	31	1	73.82	92.33

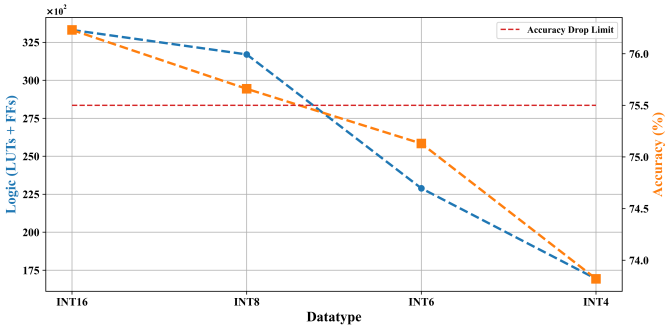


(a)

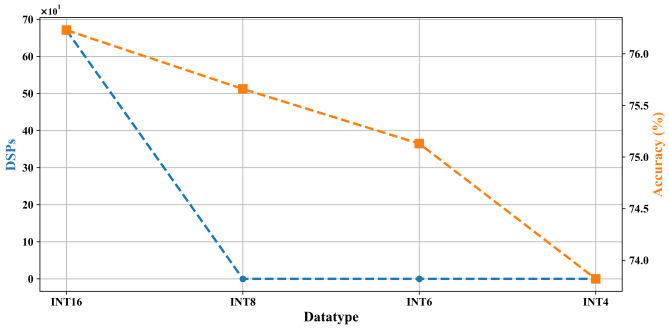


(b)

Fig. 2: Accuracy of Pruned Models with Different Sparsity vs (a) Logic and (b) DSP.



(a)



(b)

Fig. 3: Accuracy of Pruned + Quantized Models with Different Datatypes vs (a) Logic and (b) DSP.

Kintex, and Virtex families. The Artix family, in particular, is known for its compact size and low power consumption, which makes it well-suited for edge application requirements. This project aims to implement the DNN on an Artix FPGA, in line with the goals of a referenced study [3] which aims to achieve fast inference with low power consumption.

IV. Simulation Results

This section outlines the outcomes of optimizations implemented on our model using the proposed workflow. Pruning is executed through TensorFlow's constant sparsity function with a low magnitude strategy, ensuring that sparsity within each layer remains constant and parameters with lower magnitudes are zeroed out. For quantization, we employ QKeras and uniformly quantize all layers and activations. At each stage, we convert our software-level model to the HLS model using hls4ml to estimate hardware costs and timings using AMD-Xilinx Vivado HLS. Verification is conducted at two levels: high-level (CPP testbench) and co-simulation (RTL functionality in HLS). The results, validated through cycle-accurate simulations, were observed on an AMD-Xilinx Artix-7 xc7a200tfbg484 device with speed grade -3.

In Fig. 2 and Fig. 3, we illustrate the accuracy vs. hardware cost trade-offs, while Table III presents the final results of the combination of model compression techniques. Beginning with the analysis of Fig. 2, it demonstrates that structured pruning leads to a significant reduction in Logic Elements (LUTs + FFs) and the number of DSPs compared to the unpruned model, as presented in Table II. Termed as hardware-aware pruning, this approach is based on the understanding that if one of the operands for multiplication becomes zero, the HLS compiler doesn't allocate DSPs for multipliers. Another

intriguing observation from Fig. 2 is the increase in model accuracy rather than a drop, as sparsity increases. This phenomenon suggests that the model may have been overfitted and overly complex for the task at hand, and through pruning, it becomes more capable of learning better representations.

In Fig. 3, it's noticeable that even with a 75% sparsity, we maintain this level of sparsity and proceed to apply quantization. It's important to note that our tolerance for accuracy drop is limited to 1%, beyond which we cannot proceed. As depicted in Fig. 3, quantizing down to INT8 still meets the accuracy drop criteria. An intriguing aspect of using the INT8 datatype is that the HLS compiler will map all computations to LUTs, which can be advantageous for FPGAs. Additionally, for FPGAs, there's a benefit in terms of dedicated computation cores for specific datatypes. This stands in contrast to GPUs, where special cores exist for specific datatypes like INT8, and even if computations are quantized to lower bitwidths, they still map to INT8 computations.

Compared to the baseline model accuracy of 76.50%, the accuracy drop was only 0.84% with 75% sparsity and INT8 quantization. However, when compared to the direct implementation results with a reuse factor of 1 as shown in Table II, we observe a reduction in LUT utilization by 39%, FF utilization by 13%, and complete elimination of DSP utilization. It's noteworthy that for the direct implementation, we had to increase the reuse factor to 8, resulting in an interval of 8 Clock Cycles (CCs). The interval determines the number of CCs needed for the model to generate new output and receive new input. With our approach, we achieved an interval of 1, indicating a fully pipelined design and 8 times higher throughput compared to the unoptimized model.

V. Conclusion

This study has demonstrated the effectiveness of HLS and model compression techniques in optimizing the implementation of DNNs on FPGAs. The proposed workflow, which includes hardware-aware pruning and quantization, has shown significant improvements in the hardware cost. Structured pruning significantly reduced the number of Logic Elements and DSPs, making the model more suitable for real-time implementation on FPGAs. Furthermore, quantizing down to INT8 still meets the accuracy drop criteria, with the added benefit of mapping all computations to LUTs. In comparison to the baseline model, the optimized model achieved a minimal accuracy drop of 0.84% for 75% sparsity and INT8 quantization. More importantly, it reduced the LUT utilization, FF utilization, and DSP utilization by 39%, 13%, and 100% respectively. The optimized model also achieved a fully pipelined design, in which the throughput is 8 times higher than the unoptimized model. This study has made significant strides in addressing the challenges of implementing DNNs on FPGAs. Future research could delve into the development of an automated system that, given a specific DNN model and FPGA architecture, could intelligently search the design space. This system would employ an optimization algorithm to identify the most efficient implementation solution.

References

[1] "Xilinx inc, adaptable and real-time ai inference acceleration," https://www.xilinx.com/products/design-tools/vitis/vitis-ai.html.

[2] Y. Umuroglu, N. J. Fraser, G. Gambardella, M. Blott, P. Leong, M. Jahre, and K. Vissers, "Finn: A framework for fast, scalable binarized neural network inference," in *Proceedings of the 2017 ACM/SIGDA international symposium on field-programmable gate arrays*, 2017, pp. 65–74.

[3] T. Aarrestad, V. Loncar, N. Ghielmetti, M. Pierini, S. Summers, J. Ngadiuba, C. Petersson, H. Linander, Y. Iiyama, G. Di Guglielmo *et al.*, "Fast convolutional neural networks on fpgas with hls4ml," *Machine Learning: Science and Technology*, vol. 2, no. 4, p. 045015, 2021.

[4] S. Han, J. Pool, J. Tran, and W. Dally, "Learning both weights and connections for efficient neural network," *Advances in neural information processing systems*, vol. 28, 2015.

[5] "TensorFlow: Large-scale machine learning on heterogeneous systems," 2015, software available from tensorflow.org. [Online]. Available: https://www.tensorflow.org/

[6] C. Coelho, "Qkeras," *https://github.com/google/qkeras*, 2019.