Accelerating CNNs on FPGAs for Particle Energy Reconstruction

CHIJUI CHEN*, Graduate Degree Program of College of Electrical and Computer Engineering, National Yang Ming Chiao Tung University, Taiwan
YANLUN HUANG*, Department of Electrical and Computer Engineering, National Yang Ming Chiao Tung University, Taiwan
LINGCHI YANG, Institute of Electronics, National Yang Ming Chiao Tung University, Taiwan
ZIANG YIN, Department of Electrical and Computer Engineering, University of Washington, USA
PHILIP HARRIS, Laboratory for Nuclear Science, Massachusetts Institute of Technology, USA
SCOTT HAUCK, Department of Electrical and Computer Engineering, University of Washington, USA
SHIHCHIEH HSU, Department of Physics, University of Washington, USA
BOCHENG LAI, Institute of Electronics, National Yang Ming Chiao Tung University, Taiwan
KELVIN LIN†, Department of Electrical and Computer Engineering, University of Washington, USA
DYLAN RANKIN, Department of Physics and Astronomy, University of Pennsylvania, USA
ALEXANDER SCHUY, Department of Physics, University of Washington, USA

The CERN Large Hadron Collider has recently integrated deep learning (DL) models, such as DeepCalo, into their flows to enhance particle energy reconstruction. However, the challenges posed by high data generation rates, dynamic experiment conditions, and resource-intensive computations demand millisecond latency and flexible deployment of different DL models. In this paper, we present the first automated design workflow based on hls4ml to implement DeepCalo models on FPGAs. By optimizing dataflow and processing schemes in hls4ml while compressing DeepCalo with quantization-aware training, we demonstrate a fully-on-chip implementation of large CNN models while maintaining model quality. We perform a comprehensive exploration of various key design factors, summarizing our observations as useful design guidelines for future DL applications. Under realistic LHC scenarios, our results on the Xilinx Alveo U50 FPGA demonstrate an inference latency of 1.34 ms per 5 images, achieving a 5.6× speedup over the existing

*Both authors contributed equally to this paper. Model quantization and hls4ml modification were conducted by ChiJui Chen. Model optimization and FPGA exploration were performed by YanLun Huang. Both authors are the main contributors to the research framework and the manuscript writing.
†Currently at Amazon, USA

Authors' addresses: ChiJui Chen, silencekugel.ee05@nycu.edu.tw, Graduate Degree Program of College of Electrical and Computer Engineering, National Yang Ming Chiao Tung University, Hsinchu, Taiwan; YanLun Huang, yanlun172@gmail.com, Department of Electrical and Computer Engineering, National Yang Ming Chiao Tung University, Hsinchu, Taiwan; LingChi Yang, hisky1256@gmail.com, Institute of Electronics, National Yang Ming Chiao Tung University, Hsinchu, Taiwan; Ziang Yin, lostecho@uw.edu, Department of Electrical and Computer Engineering, University of Washington, Seattle, USA; Philip Harris, pcharris@mit.edu, Laboratory for Nuclear Science, Massachusetts Institute of Technology, MA 02139, USA; Scott Hauck, hauck@uw.edu, Department of Electrical and Computer Engineering, University of Washington, WA 98195, Seattle, USA; ShihChieh Hsu, schsu@uw.edu, Department of Physics, University of Washington, WA 98195, Seattle, USA; BoCheng Lai, bclai@nycu.edu.tw, Institute of Electronics, National Yang Ming Chiao Tung University, Hsinchu, Taiwan; Kelvin Lin, kelvin.lin1@gmail.com, Department of Electrical and Computer Engineering, University of Washington, WA 98195, Seattle, USA; Dylan Rankin, dsrankin@sas.upenn.edu, Department of Physics and Astronomy, University of Pennsylvania, 209 South 33rd Street, Philadelphia, USA; Alexander Schuy, schuya@uw.edu, Department of Physics, University of Washington, WA 98195, USA.

Manuscript submitted to ACM

GPU-based system. At a batch size of one, the image-only model demonstrates a latency of 0.443 ms, while the full model exhibits a latency of 1.34 ms, meeting the latency requirement of the Level-1 Trigger in particle experiments. Compared to the Ryzen-5600H CPU and Tesla V100 GPU, we achieve speedups of 14.1× and 7.9×, respectively.

ACM Reference Format:
ChiJui Chen, YanLun Huang, LingChi Yang, Ziang Yin, Philip Harris, Scott Hauck, ShihChieh Hsu, BoCheng Lai, Kelvin Lin, Dylan Rankin, and Alexander Schuy. 2023. Accelerating CNNs on FPGAs for Particle Energy Reconstruction. 1, 1 (August 2023), 29 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Data analysis is an important element of particle physics [1]. Modern experiments increasingly rely on machine learning (ML) for timely and efficient analysis of large volumes of data. One such example is the Large Hadron Collider (LHC) [2] at CERN [3], where proton-proton collisions occur every 25 ns and are recorded by detectors with millions of channels. The signals from these detectors are then processed by ML algorithms to filter uninteresting data in real-time, despite the large data rate of 100 TB/s [4].

With advances in deep learning (DL) and processor architecture in recent years, LHC experiments have adopted DL to improve the quality of data analysis [5]. DeepCalo [6] is a Keras-based [7] DL design specifically created for simulation data from ATLAS [8], one of two general-purpose detectors at the LHC. It allows users to build, train, tune, and test convolutional neural networks (CNNs) that reconstruct the energy of particles such as electrons and protons [9] [10]. Unlike the conventional analysis approach of boosted decision trees (BDTs), which are trained only on derived scalar variables [11], DeepCalo directly processes images created from the electromagnetic calorimeter (ECAL). Compared to BDTs, DeepCalo improves the energy reconstruction accuracy by 11.9% - 20.9% for electrons, and 17.7% - 27.5% for photons, depending on the energy range and detector region [12]. The current version of DeepCalo supports two scenarios: a full model of 3.6M parameters which combines images, scalar variables, and track vectors; and an image-only model of 1.8M parameters that only processes images.

Although DeepCalo performs well in energy reconstruction, its resource-intensive computation makes on-line inference challenging due to the strict latency requirements of the LHC. The LHC event-selection system consists of three tiers: level-1 trigger (L1T), high-level trigger (HLT), and offline reconstruction [13] [14]. L1T and HLT are online computing systems that operate at 40 MHz (100 kHz) with a latency of ~1 μs (~10 ms) [15] [16]. The L1T is implemented using specialized electronics, while the HLT utilizes software running on a compute farm. According to [17], even for HLT, it still requires six GPU servers with a bandwidth of 24 Gb/s to run DeepCalo. Therefore it is more practical to handle this regression task offline. Furthermore, the future High-Luminosity LHC project plans to increase the beam intensity by 5× to 7× by 2027 [18] [19], which would make it extremely difficult, if not impossible, for CPU/GPU solutions to meet the stringent processing requirement.

Field Programmable Gate Arrays (FPGAs) enable customized data processing logic and have been broadly adopted to attain highly parallel dataflow processing with short latencies. The LHC has deployed FPGAs for online inference data analysis [20] [21] [22] [23] [24] [25], and facilitated the design with hls4ml [26]. hls4ml is a high-level synthesis (HLS) tool that converts high-level descriptions of ML algorithms into efficient FPGA implementations. However, the current hls4ml framework has limited support for converting large-scale models like DeepCalo. This is because hls4ml implements a fully on-chip dataflow architecture in order to avoid long-latency DRAM accesses. This approach needs to implement all the ML layers on an FPGA and therefore poses stringent constraints on the size and complexity of the model. Moreover, hls4ml's stream-based dataflow is constrained by the channel size, since resource consumption

increases drastically when the channel size increases, and increased channel size is common in most CNNs. Also, quantization has been an effective technique to reduce design complexity. However, how to strike the balance between quantization errors and precision bit-widths has become a serious design concern when dealing with large models.

In this paper, we present the first fully-automated design and optimization workflow based on hls4ml to implement DeepCalo models on FPGAs. We not only perform a comprehensive exploration of various key design factors but also propose a design that attains shorter latency (<1ms) than solutions on CPUs and GPUs. We extend the DeepCalo framework and integrate QKeras layers to perform quantization-aware training (QAT), which is crucial for minimizing resource consumption and maximizing model performance [27][28]. With a highly efficient streaming dataflow and optimized architectures of most neural network layers in hls4ml, we are able to support the automatic conversion of large-scale CNNs to HLS, and then to an implementation on a Xilinx Alveo U50 FPGA board [29]. We further explore the rounding strategies used in hardware to reduce the quantization error when transferring the model to FPGAs, in order to obtain a good balance between resource utilization and accuracy. Considering real LHC scenarios, the image-only model has an inference latency of 1.34 ms per 5 images, achieving a 5.6× speedup over the existing GPU-based system. Compared to the Ryzen-5600H CPU [30] and Tesla V100 GPU [31], the implementation of the image-only (full) model on FPGA shows up to **14.1×(9.7×)** and **7.9×(5.3×)** speedups respectively.

This paper is structured as follows: Section 2 provides background knowledge, including CNN acceleration in other frameworks and in hls4ml for FPGAs, as well as an overview of DeepCalo and the existing GPU-based data reconstruction system. Section 3 discusses the hardware implementation and optimization of DeepCalo models using HLS. Section 4 details the model compression using QAT and compares the resource usage and performance of different rounding methods in HLS. The experimental results of DeepCalo are presented in Section 5. Conclusions are given in Section 6.

2 BACKGROUND

This section provides essential background knowledge to contextualize our research. We begin by introducing various frameworks designed for accelerating CNNs on FPGAs, assessing their respective strengths and limitations. This exploration leads us to select hls4ml as the framework upon which to build our study. Subsequently, we describe previous CNN implementations within hls4ml, aiming to extend the advantages and integrate larger CNNs in hls4ml. In addition, we introduce DeepCalo. Lastly, we introduce an existing GPU-based system designed for data reconstruction, including the results of running DeepCalo on those GPUs.

2.1 Related Works to Accelerate CNNs on FPGAs

Various design frameworks offer flexibility for users to customize their ML models on FPGAs. CNN accelerators on FPGAs can be categorized into two types: customized dataflow processing and generic processing. Customized dataflow processing allows users to design and optimize the dataflow architecture specific to their models. This approach provides low latency and high throughput by leveraging parallel processing and fully on-chip architectures. As a result, it is suitable for smaller neural networks and platforms that lack computing units. Several frameworks exemplify this approach, including hls4ml and FINN [32] [33]. FINN is an open-source framework to explore deep neural network inference on FPGAs, targeting quantized neural networks with dataflow-style architectures. Notably, FINN has demonstrated an on-chip implementation of large CNNs such as ResNet50 for Xilinx Alveo boards, which utilizes ultra-low bit-width quantization and achieves millisecond-level latencies.

In contrast, generic processing approaches provide standardized and pre-optimized frameworks or libraries for CNN acceleration. Eyeriss [34] is a well-known framework that employs a row stationary (RS) architecture, which focuses on optimizing both computation and memory access patterns. FlexCNN [35] is a framework that adopts a reconfigurable systolic array architecture, enabling it to adapt to different CNN models and layer sizes. Vitis AI [36] is a platform for comprehensive AI inference developed by Xilinx. It consists of optimized deep learning processor unit (DPU) cores, tools, libraries, and example designs. Although the approaches mentioned above are flexible and consume lower resources, it requires frequent transfers of weights and feature maps between FPGA on-chip memory (BRAM) and external memory (DDR/HBM), which is challenging to perform for real-time inference in the L1T. Moreover, generic processing is restricted to specific layers and CNN topologies, especially when deploying models with custom layers like DeepCalo.

Despite several frameworks featuring ultra-low latency and power consumption, in this paper, we choose hls4ml due to its close relationship with the high-energy particle physics field and the ability to convert mixed-precision models from QKeras. We extend the hls4ml library to support large CNN conversion while staying fully on-chip, as well as generating a dataflow architecture. In addition, we offer a complete quantization solution that can support other models, in which the quantization error can be reduced and tested during the early design stage.

2.2 Previous CNNs Implementations in hls4ml

FPGAs enable customized data processing logic and have been broadly adopted to attain highly parallel dataflow processing with short latency for the inference of ML models [37] [38] [39]. Previous studies in hls4ml have primarily focused on achieving millisecond-latency inference for CNNs on FPGAs. The initial work by Smith et al. [23] introduced support for streaming-based CNNs in hls4ml. This process removed allocating resources to monitor the location of elements in the sliding window or image corners management by computing and encoding positions as binary masks in advance. This strategy allows for the efficient retrieval of correct data in sliding windows through cooperation with buffered streams. Building upon this, Ref. [40] enhanced the stream-based approach with an optimized linebuffer architecture for real-time semantic segmentation tasks. The accelerator was compressed with automatic heterogeneous QAT and a filter ablation procedure, achieving a latency of 4.9 ms per image. These two convolution implementations, known as "encoded" and "linebuffer" are currently available options in hls4ml. The authors of [40] proposed using linebuffers which utilize shift registers to record previously seen pixels, thus reducing the memory needed to store duplicated pixels. For an image of size $H \times W$, with a convolution kernel of size $K \times L$, the line buffer allocates $K - 1$ buffers (chain of shift registers) of depth W for the rows of the image, while the "encoded" implementation allocates K^2 buffers of depth $K \times (W - K + 1)$ for the elements in the sliding input window.

In our work, we build upon the "linebuffer" scheme and focus on optimizing the dataflow of the streams along the channels. By leveraging the advantages highlighted in [40], we aim to enhance the efficiency and scalability of larger CNNs in hls4ml.

2.3 DeepCalo: Deep Learning Framework for ATLAS Data

Deepcalo is a deep learning framework for training CNNs on ATLAS simulation data, typically for energy reconstruction of electrons and photons. Enhancing the reconstruction quality can improve the accuracy of data analyses, such as the substantial Higgs boson decay channels [41]. To accurately reconstruct electron and photon energy, the full model processes 3 input sources simultaneously: images, scalar variables, and track vectors. The image pixels corresponding to the electromagnetic calorimeter (ECAL) cells were constructed using the Monte Carlo method [42], which is commonly

Table 1. Output shape, number of parameters, floating-point operations, and energy consumption of each layer in the full model. The values of each term after summation are also listed.

(a) CNN (image-only model)

Layer	Output shape	Parameters	MFLOPs	Energy(nJ)
Input Images	(56, 11, 4)	-	-	-
Upsampling2D	(56, 55, 4)	-	-	1,171.57
Conv2D1	(56, 55, 16)	1,681	9.856	99,583.22
MaxPool1	(28, 27, 16)	-	0.049	-
Conv2D2	(28, 27, 32)	4,769	6.967	69,015.96
Conv2D3	(28, 27, 32)	9,377	13.935	92,021.28
MaxPool2	(14, 13, 32)	-	0.024	-
Conv2D4	(14, 13, 64)	18,753	6.709	33,229.91
Conv2D5	(14, 13, 64)	37,185	13.418	44,306.54
MaxPool3	(7, 6, 64)	-	0.012	-
Conv2D6	(7, 6, 128)	74,369	6.193	15,336.88
Conv2D7	(7, 6, 128)	148,097	12.386	20,449.18
MaxPool4	(3, 3, 128)	-	0.005	-
Conv2D8	(3, 3, 256)	296,193	5.308	6,572.95
Conv2D9	(3, 3, 256)	591,105	10.617	8,763,94
Flatten	(2304)	-	-	4,381.97
Dense1	(256)	590,849	1.180	4,868.86
Dense2	(256)	66,561	0.131	973.78
Dense3	(1)	257	0.000512	842.35
Total	-	1,839,196	86.792	401,518

(b) Scalar Net, Track Net, and FiLM Generator

Layer	Output shape	Parameters	MFLOPs	Energy(nJ)
Scalar Net				
Input Scalar Variables	(15)	-	-	-
Dense	(256)	4,865	0.008	494.5
Track Net				
Input Track Vectors	(88, 6)	-	-	-
Time Distributed	(88, 128)	18,178	0.031	251.05
Sum1D	(128)	-	-	21,422.94
Dense1	(128)	16,897	0.033	486.88
Dense2	(128)	16,897	0.033	486.88
FiLM Generator(Connecting the Track and Scalar Net)				
Concatenate	(384)	-	-	730.33
Dense1	(512)	198,675	0.393	1,704.1
Dense2	(1024)	528,385	1.049	2,921.31
Dense3	(992)	1,016,800	2.032	1,382,706.30
FiLM1	(56, 55, 16)	-	0.09856	95,672.92
FiLM2	(28, 27, 32)	-	0.048384	47,958.18
FiLM3	(14, 13, 64)	-	0.023296	24,100.81
FiLM4	(7, 6, 128)	-	0.010752	12,172.13
FiLM5	(3, 3, 256)	-	0.004608	6,816.4
Total	-	1,800,697	3.7646	1,597,924.28
Total (Include CNN)	-	3,639,893	90.5566	2,038,838.69

used in particle physics to model the behavior of particles in detectors. The full model contains 62 layers, leading to 3.64 million parameters. There is also an image-only model which takes only ECAL images and passes them through the CNN layers. In the following paragraphs, we describe the model architecture in detail.

The structure of the full model is illustrated in Figure 1. The ECAL CNN (referred to as the image-only model) contributes the most discrimination power in the system. It follows a VGG-style [43] architecture that comprises five 2D convolutional blocks and three dense blocks. Each convolutional and dense layer is followed by a batch normalization layer and a rectified linear unit (ReLU) activation function. The initial block is responsible for upsampling the input pixels to a square-like shape, consisting of a Upsampling2D layer, and a convolutional layer with a kernel size of 5×5. All subsequent blocks have the same structure: they begin with a 2×2 max-pooling layer, followed by two sets of a single convolutional layer with a 3×3 kernel size. The convolutional layers in the l^{th} block have $16l$ filters. In the initial two dense blocks, each dense layer consists of 256 neurons, while the last dense block includes a dense layer with a single neuron that produces the final regression outcome. ReLU is employed as the final activation function.

The scalar variables and track vectors undergo processing by their own individual submodels. The Scalar Net consists of a dense layer with 256 neurons, whereas the Track Net has a TimeDistributed layer that applies a dense net to process each track vector associated with an event individually. The resulting output for each track vector is summed up and fed into another dense net for further processing. Moreover, to incorporate the effects of additional input variables, feature-wise linear modulation (FiLM) layers [44] are introduced into CNN computation. Specifically, these layers are positioned after the initial convolutional layer in each 2D convolutional block. Through this approach, the behavior of

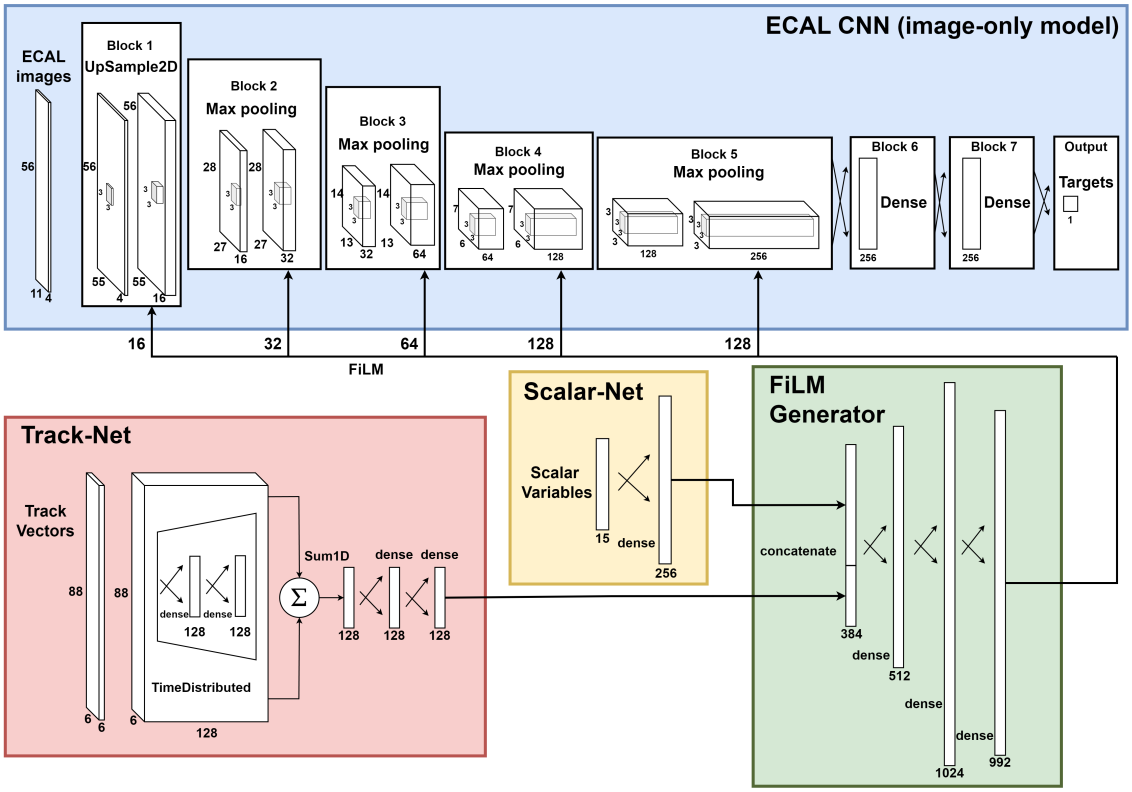


Fig. 1. Structure of the Deepcalo *full model*. In addition to the main CNN(*image-only model*), the *full model* contains additional three submodels: the Track Net, the Scalar Net, and the FiLM Generator.

the CNN is dynamically altered in an adaptive manner as a function of all three input variables. The outputs of the Scalar and Track Net are concatenated and fed into the FiLM generator, which is a basic fully-connected network. This generator is responsible for generating scaling and shifting factors for the FiLM layers, which are utilized to modify the feature maps of the CNN.

The detailed parameters of output shape, number of floating-point operations (FLOPs), and parameters for each layer are listed in Table 3. In addition, an estimate of the per-layer energy consumption is demonstrated, which was estimated using QTools [45] within the context of a 45 nm processor. Despite the last dense layer in the FiLM generator containing the most weights, the convolutional layers in the main CNN exhibit considerably higher levels of energy consumption and FLOPs due to the substantially larger number of multiply-accumulate (MAC) operations performed.

2.4 GPU-based Hardware Acceleration in Data Reconstruction Workflow

A comprehensive exploration of the adoption of GPU-based hardware acceleration for data reconstruction in the LHC workflow was presented in [17]. This was done by extending the Services for Optimized Network Inference on Coprocessors (SONIC) framework in the Compact Muon Solenoid (CMS) experiment [46]. The existing CPU-based workflow is reconfigured, and algorithms are transferred to GPUs without disruption. The study considered three DL-based reconstruction algorithms of varying scales, including the image-only model of DeepCalo. Instead of operating

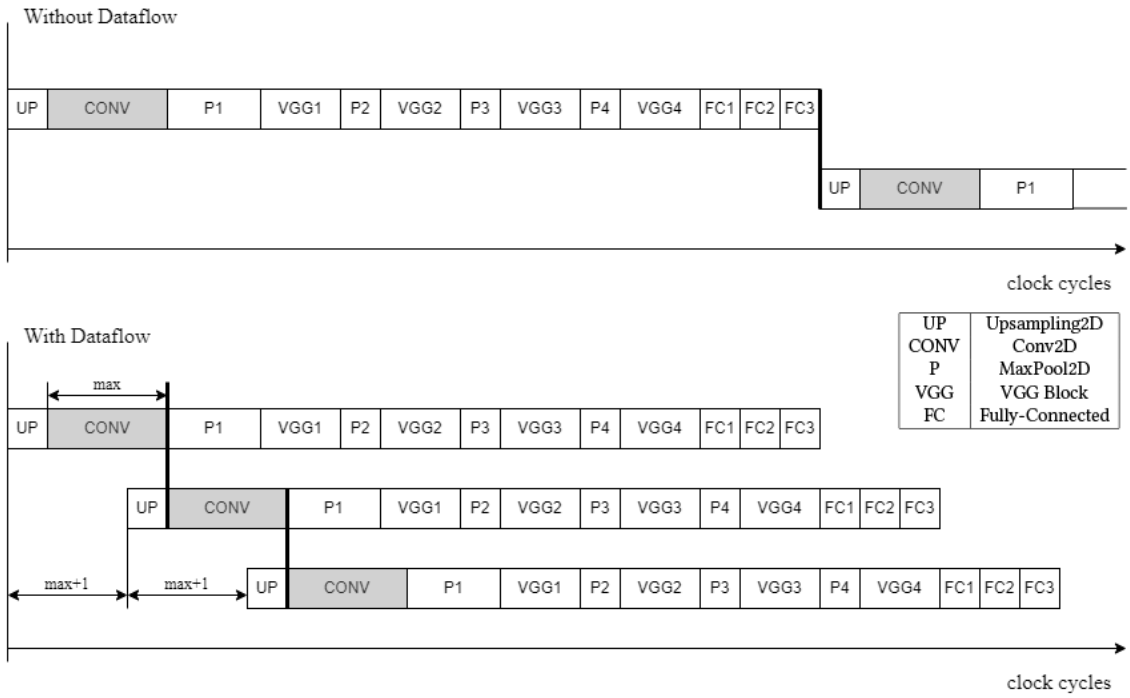


Fig. 2. Comparison of the image-only model operation stages with (lower) and without (upper) dataflow pipeline. In the lower subfigure, different operation stages can be executed at the same time to enhance the processing throughput.

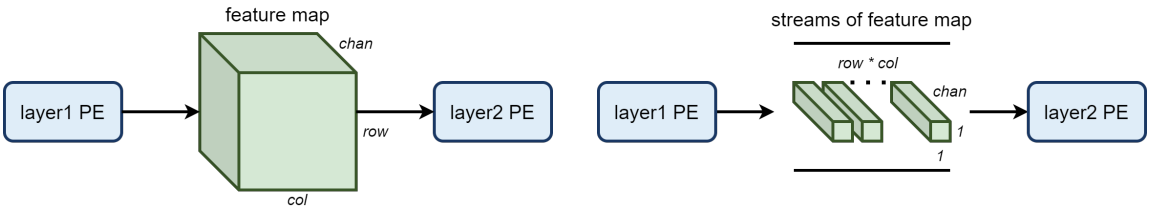
in the HLT, the model was deployed as a service on GPU coprocessors for offline reconstruction due to the impractical high data rate. Considering realistic LHC scenarios with events involving 5 electrons, the inference latency per event was reduced from 75 ms to 1.5 ms compared to the CPU-based implementation.

3 A STREAM-BASED ARCHITECTURE OF DEEPCALO ON FPGA

As mentioned in previous sections, DeepCalo needs to be implemented as a fully-on-chip and dataflow architecture to attain high throughput and meet the stringent latency constraints of the LHC. In this section, we will elaborate on the design of the stream-based architecture and the optimization in different layers.

3.1 Stream-based Architecture

To realize the DeepCalo data flow architecture, the network layers are pipelined using *pragma HLS Dataflow* from Vivado HLS [47]. As illustrated in Figure 2, different processing stages (layers) are implemented in a pipeline manner and can be executed simultaneously. Data transmission schemes are crucial in a dataflow architecture. hls4ml provides two methods to transmit data between layers: **array-based** and **stream-based**, resulting in distinct processing structures as illustrated in Figure 3. The array-based scheme stores the complete feature map in an array and transmits it to the next layer only after the computation is finished. While this approach enables maximum parallel processing, it can result in large storage requirements and potentially increased timing overhead. In contrast, the stream-based approach utilizes FIFOs (First In First Out buffers) for connections between stages, eliminating extra data management and improving



(a) array-based (b) stream-based

Fig. 3. Comparison of (a) array-based and (b) stream-based data transmission schemes in **hls4ml**.

resource utilization. Furthermore, it efficiently supports the sequential pipeline within a dataflow structure. Therefore, our design in this paper adopts the stream-based architecture.

To facilitate the later discussion, we use symbols H, W, and N to respectively denote the height, width, and number of channels for an input feature map. We also employ the variables *n_in* and *n_out* to represent the corresponding input and output dimensions of a dense layer.

3.2 Analysis of Different Stream Types

Applying different types of streams has an impact on various aspects, such as resource usage, latency, and the architecture of corresponding processing elements (PEs). Three of the most common streaming approaches in Vivado HLS are discussed in this paper: stream-of-struct, single-value stream (hereafter called single-stream), and array of single-value streams (hereafter called array-of-streams). Figure 4 shows the flows of three types of streams from one layer to another, together with their corresponding HLS C++ codes. The default stream type utilized in hls4ml is stream-of-struct, which transfers an entire struct containing an array of size N. In single-stream, one pixel is transferred per cycle, whereas array-of-streams involve multiple parallel streams. In the following paragraphs, we will compare them and introduce suitable application scenarios.

In stream-of-struct, there are two steps involved. First, *pragma HLS data_pack* [48] is applied to partition and reshape the array into a unified long vector, as shown in the bottom left side of Figure 4. Second, as the subsequent layer processes the entire channel concurrently, Vivado HLS partitions the stream into N individual streams. This allows the PEs to fetch N input data in a single cycle. However, this approach increases the circuit complexity.

The array-of-streams is composed of N parallel streams, resulting in N FIFOs, and resembles the stream-of-struct architecture. The difference is that there is no requirement to first apply *pragma HLS data_pack* to expand the bandwidth. This greatly simplifies the architecture and reduces the compile time in Vivado HLS.

The single-stream operates with a single FIFO and conducts data transmission in a serial manner. As a result, the need for loop unrolling or array partitioning to facilitate parallel computation is eliminated, making it more straightforward to implement.

The current hls4ml performs effectively when the channel size is small, allowing low latency in CNN models. However, when dealing with larger CNN models, synthesizing the design becomes unpractical due to the longer compilation time and the increased network complexity. The resource utilization experiment involving different streams is discussed in detail in Section 5. Based on our observation, models with convolutional layers that have more than **64** filters would encounter this issue. To address these limitations, we switched to single-stream and array-of-streams

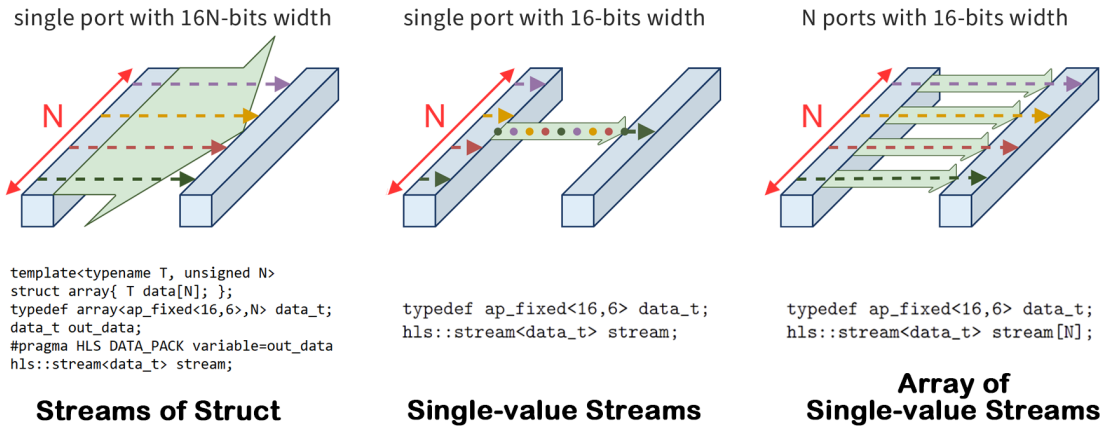


Fig. 4. Illustration of different stream types: Streams of Struct (left), Single-Value Streams (middle), and Array of Single-Value Streams (right). We assume that all pixels are 16 bits and N pixels (the entire channel) are transferred. The streams flow from the left-hand side (source layer) to the right-hand side (destination layer). HLS C++ codes are included to demonstrate stream creation.

Algorithm 1 The definition of *data_transfer_in/out* variables in hls4ml.

Input: positive integers: ***t*** denotes the threshold value, ***in*** denotes the input channel size, and ***out*** denotes the output channel size.
Output: positive integers: ***data_transfer_in*** and ***data_transfer_out*** variables.

```
Get t, in, out
if in ≤ t then
    data_transfer_in ← in
else
    data_transfer_in ← 1
end if
if out ≤ t then
    data_transfer_out ← out
else
    data_transfer_out ← 1
end if
Get data_transfer_in, data_transfer_out
Stop
```

for data transmission in the hls4ml framework. This change required the reimplementation of the hardware mapping mechanism and most HLS C++ layers.

3.3 Choosing the Proper Stream Type for Network Layers

In DeepCalo, downsampling leads to a decrease in spatial dimensions ($H \times W$) as the number of channels (N) increases. In the single-stream approach, it takes $H \times W \times N$ cycles to process all elements. However, in the array-of-streams approach, the required cycles are reduced to $H \times W$, saving a factor of N cycles. As a result, when the channel size is below a certain threshold (default 64), an array-of-streams strategy is used to shorten the latency, even though it

Algorithm 2 The switch function for the data transfer method in Conv2D.

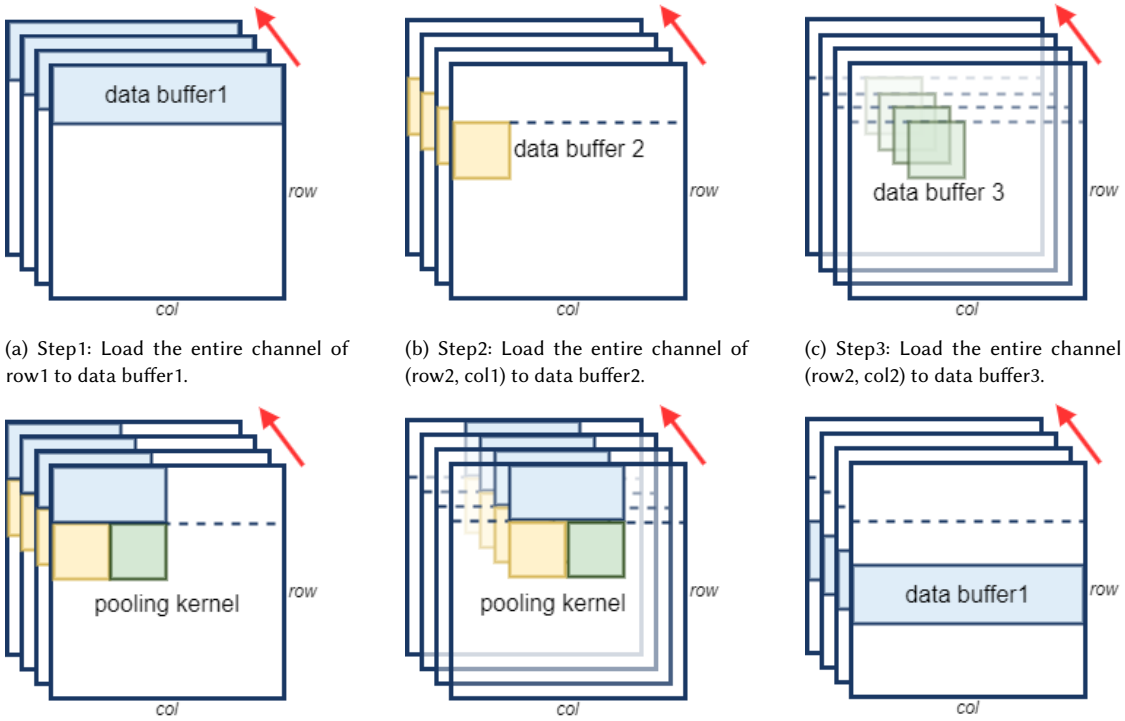
Input: a positive integer ***data_in*** denotes the data_transfer_in variable, and a positive integer ***data_out*** denotes the data_transfer_out variable
Output: ***method*** denotes the applied data transfer function
- single-stream to single-stream, denote as *S2S*
- single-stream to array-of-streams, denote as *S2A*
- array-of-streams to single-stream, denote as *A2S*
- array-of-streams to array-of-streams, denote as *A2A*

Get *data_in*, *data_out*
if *data_in* == 1 **then**
  if *data_out* == 1 **then**
    method ← *S2S*
  else
    method ← *S2A*
  end if
else
  if *data_out* == 1 **then**
    method ← *A2S*
  else
    method ← *A2A*
  end if
end if
Get *method*
Stop

requires more resources such as Block RAMs (BRAMs) for FIFOs, digital signal processors (DSPs), and lookup tables (LUTs) for parallel computation. But for larger channel sizes, using array-of-streams leads to a complicated hardware design, and the benefit of further reducing transmission time is not significant, as the input feature maps have already been downsampled multiple times. In those cases, a single-stream strategy is used instead. The *pragma HLS pipeline* is applied for the concurrent execution of operations, rather than using *pragma HLS UNROLL* [49] to create duplicate PEs.

To optimize the design for the available hardware resources, a switch function is introduced to hls4ml. It automatically converts data types between single-stream and array-of-streams based on channel size. The threshold value for switching can be adjusted by the user based on their requirements. This feature is facilitated by adding two new variables in all layers: *data_transfer_in and data_transfer_out*, which record the input-stream and output-stream channel sizes respectively. The definition of these two variables is depicted in Algorithm 1.

Once the threshold value is set, hls4ml fills in the values of *data_transfer_in and data_transfer_out* automatically. If the number of input(output) channels exceeds the threshold value, the value of *data_transfer_in*(*and data_transfer_out*) will be set to 1, indicating that the single-stream will be applied. Otherwise, the value of *data_transfer_in*(*and data_transfer_out*) will be equal to the input(output) channel size, and array-of-streams will be employed. The implementation of the switch function is demonstrated in Algorithm 2. After the hls4ml conversion, these two values are defined in the configuration file and used by Vivado HLS to generate the corresponding hardware design. By optimizing the use of streams, the implementation process becomes more efficient and effective in terms of both latency and resource consumption.



(a) Step1: Load the entire channel of row1 to data buffer1.

(b) Step2: Load the entire channel of (row2, col1) to data buffer2.

(c) Step3: Load the entire channel of (row2, col2) to data buffer3.

(d) Step4: Channel-wise max-pool in the pooling kernel.

(e) Step5: Shift the pooling kernel two strides to the right.

(f) Step6: Move on to the next row and repeat steps 1-5.

Fig. 5. Step-by-Step illustration of a 2×2 Max-pooling layer process.

3.4 Architecture Optimizations for Layers

Apart from the advantages of reorganizing the stream communication itself, the new transmission pattern allows us to optimize the processing units. For instance, long compilation and synthesis duration are observed in dense layers, which are attributed to the calculation pattern and weight arrangement. Another example is the max-pooling layers, where Initiation Interval (II) violations are encountered when the sequential transmission is applied. In the following subsections, we will analyze the causes of these issues and present our solutions to address them.

3.4.1 Max-pooling Layer Optimization. Similar to convolutional layers, hls4ml also provides a "linebuffer" option in max-pooling layers. Although the linebuffer scheme offers flexibility in terms of pooling sizes and strides, it requires numerous shift registers and sliding window buffers to minimize processing cycles. In addition, using single-stream results in an initiation interval (II) violation on the input due to a mismatch between the original design, which processes the entire channel concurrently. This would lead to a routing congestion problem and force the compiler to reduce the clock frequency on FPGAs.

To overcome these limitations, an optimized design approach has been developed as presented in Figure 5. The design focuses on the most common max-pooling layers with 2×2 pooling sizes and strides of two. Since there is no overlapping between kernels, data could be stored in separate buffers. This approach does not require shift-register

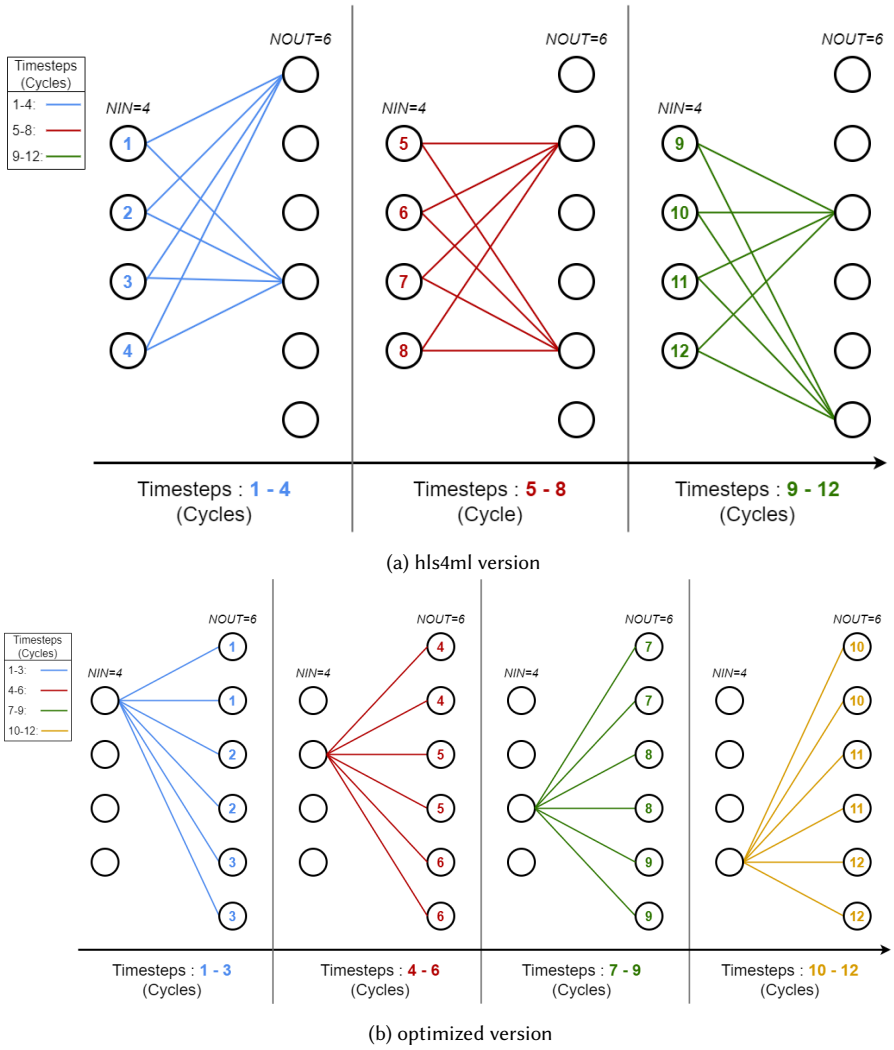
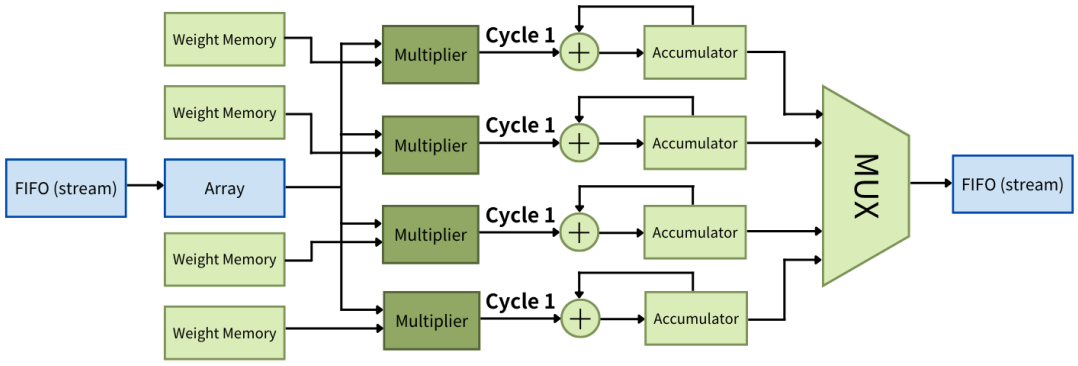
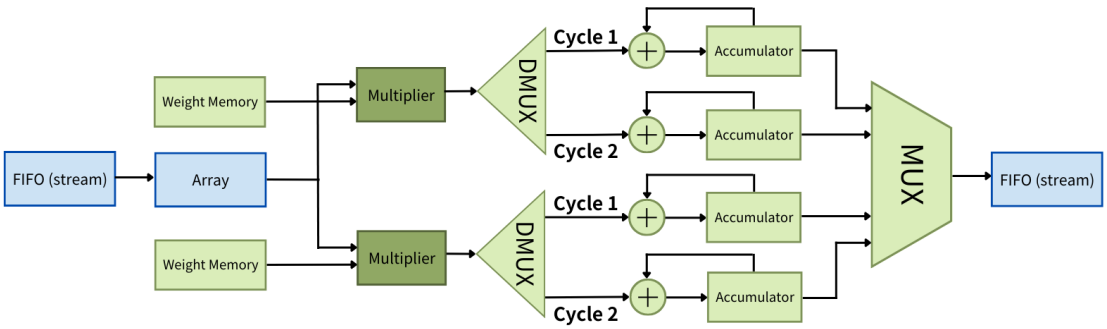


Fig. 6. Comparison of calculation patterns in dense layer: (a) hls4ml version and (b) optimized version. Both layers have 4 input neurons and 6 output neurons, and use 2 DSPs.

linebuffers and allows data to be retrieved every cycle, therefore solving the above problems. Array-of-streams is applied in the input, so the below processes are all performed across the entire channel. Three buffers are utilized: Initially, with a pipelined process, $W + 1$ cycles are spent to store the first row in the first buffer. Next, we move to the second row and access the data from the first column, storing them in the second buffer. Finally, data in the right column are assigned in the third buffer. At this moment, all the values required for computing pooling have been gathered, and the pooling result can be calculated and sent into the output FIFOs channel-wise. For the remaining pixels in the second row, we shift the pooling kernel two steps (stride 2) to the right and reuse the second and third buffers. The process is continued until we reach the end of the second row. Then, we repeat the process by storing the third row of input data in the first buffer again and continue until we reach the final pixel.



(a) 4 DSPs



(b) 2 DSPs

Fig. 7. Comparison of the different value of *reuse factor* in the optimized stream-based version of dense layer: (a) a dense layer with 4 DSPs and (b) a dense layer with 2 DSPs.

3.4.2 Dense Layer Optimization. hls4ml includes a dedicated dense layer, which offers a parameter known as *reuse factor* for reuse of multiplier. Adjusting the value of *reuse factor* can, in turn, affect the number of computation PEs utilized. Therefore, users can adjust the number of DSPs in the circuit to match the specific FPGA platform. Figure 6a provides an overview of how the original hls4ml algorithm works. In the blue section, the first input is selected and two DSPs are utilized to perform the multiplications and accumulations (MACs) to apply that input to the first and fourth output. The same process is repeated for the second, third, and fourth inputs, at which point the first and fourth output are completed. Once the computations in the blue section are done, the circuit starts to process the red section. At that time, the first input is selected again to perform MACs related to the second and fifth outputs, followed by the same process for the remaining inputs. Finally, in the green section, the inputs are reused again to perform MACs associated with the third and sixth outputs. Due to the complex index arrangement of weights and the repeated reuse of inputs, synthesizing the circuit becomes time-consuming and requires more resources, especially when dealing with larger input and output dimensions. DeepCalo models, which include multiple large-scale dense layers, further amplify the synthesis time and complexity of the circuits. Therefore, it was necessary to adjust the algorithm.

A revised approach is depicted in Figure 6b. The algorithm starts by selecting the first input and performing all associated MACs, as indicated by the blue section. With two DSPs utilized, it takes three cycles to complete the

computations. Next, the second input is selected and performs all relevant MACs, depicted in the red section. This process is repeated for the remaining inputs, as the green section and the yellow section illustrate. With this approach, the circuit becomes simpler since the accumulator index is adjacent, and each input is used only once. Additionally, the weight array is transposed to ensure alignment with its corresponding input. Based on our experience, the overall synthesis time is reduced from the scale of hours to minutes.

We also provide a mechanism to adjust the number of DSPs by utilizing the *reuse factor*. In the hls4ml dense layer, both the input and output are considered, so the *reuse factor* is a factor of $n\_in \times n\_out$. However, in the optimized dense layer, the *reuse factor* is solely determined by the number of output neurons, so it is a factor of *n_out*. For instance, the *reuse factor* is 12 in Figure 6a, and is 3 in Figure 6b. Figure 7a illustrates four DSPs working in parallel. Figure 7b shows only two DSPs working in parallel, resulting in a latency that is twice that of Figure 7a.

4 EXPLORATION AND REFINEMENT OF QUANTIZATION

In the current design flow, quantization errors could arise due to converting QKeras models to HLS ones. These errors occur because the original floating-point computations are replaced by restricted-width fixed-point operations that are significantly more efficient in FPGA logic. To fix the problem, quantization-aware training (QAT) is adopted so that the model has weights in fixed point. Compared to a post-training quantization (PTQ) model, even if a QAT model is hard to train and needs a large dataset to fine-tune, it outperforms in low bit-width and has a lower quantization error. However, this can result in rounding or overflow of the arithmetic computation. It is essential to identify these errors in the early stage to ensure efficiency in the long train-to-inference workflow. This section presents a detailed analysis of the quantization process and proposes solutions to mitigate or eliminate quantization errors. The goal is to maintain the accuracy and reliability of the converted HLS models.

4.1 Impact of Insufficient Bit-width in Accumulator

4.1.1 Determination of Bit-width in Accumulator. Dense and convolution layers consists mainly of multiplier-accumulator (MAC) units, followed by units for activation functions, as shown in Figure 8a and 8b. In QKeras, quantizers are used as constraints to tune the bit-width of weights, bias, and activations to a specific distribution. This is done using the quantization formula listed in Equation (1). BW_{total} and BW_{int} respectively denote the total bit-width and bit-width in the integer part that we used to represent the fixed-point number from the input. FP_q and FP_{nq} denote the quantized and non-quantized floating point number. In Equation (1), FP_q will be scaled and rounded to an integer, clipping the number so that the integer will be saturated to the maximum or minimum that can be represented according to the bit width, re-scaling the number to a fixed-point number. Values will be quantized during training so that the training can accurately adapt to any effects introduced by quantization. However, during the accumulation phase of matrix multiplication, it's not feasible to apply quantization to the data. This is due to the nature of the Keras backend and CUDA, which perform the matrix multiplication operations on the GPU. These operations are essentially 'wrapped up' in the backend, meaning we cannot easily insert a quantizer during this process when training the model.

The accumulation result is quantized before the activation function. Consequently, this necessitates the pre-determination of the range of values in the accumulator before initiating the inference process. After converting from Keras floating-point to HLS fixed-point format, one way to choose the best precision of the accumulator is to test the model with various bit-widths until the error is acceptable. However, this is very time-consuming for searching. We adopted a more efficient approach in which we estimate the required bit-width by using the number of additions involved in producing a result. The relation between accumulation bit-width and additions is formulated in Equations

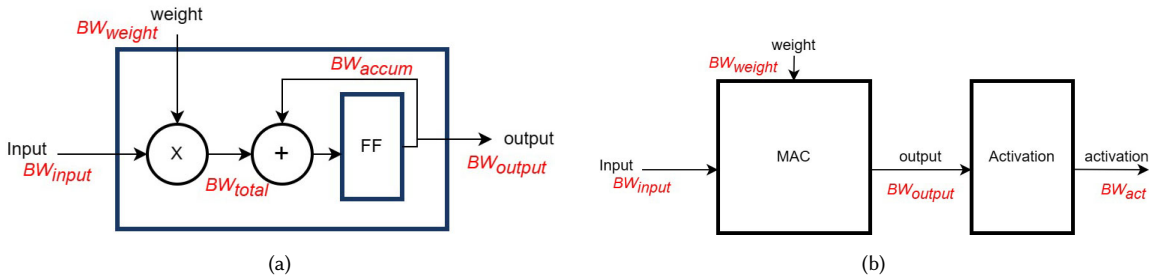


Fig. 8. Computational graphs of (a) a multiplier-accumulator (MAC) unit (b) a MAC unit followed by an activation-computing unit. Different types of precision in various domains are indicated in red font.

(2) to (4). BW_{acc} denotes the required bit-width in the accumulator, N_{acc} denotes how many additions are performed in the computation, and $FP_{q,N_{acc}}$ denotes the quantized floating-point number after N_{acc} additions. Equation (2) sets the limit of FP_q, and Equation (3) sets the limit of $FP_{q,N_{acc}}$. Based on these two bounding conditions, we can estimate the bit-width of the accumulator (BW_{acc}) using Equation (4).

$$FP_q = \frac{2^{BW_{int}}}{2^{BW_{total}}} \cdot Clip(Round(FP_{nq} \cdot \frac{2^{BW_{total}}}{2^{BW_{int}}})) \tag{1}$$

$$-2^{BW_{total}-1} \leq FP_q \leq 2^{BW_{total}-1} - 1 \tag{2}$$

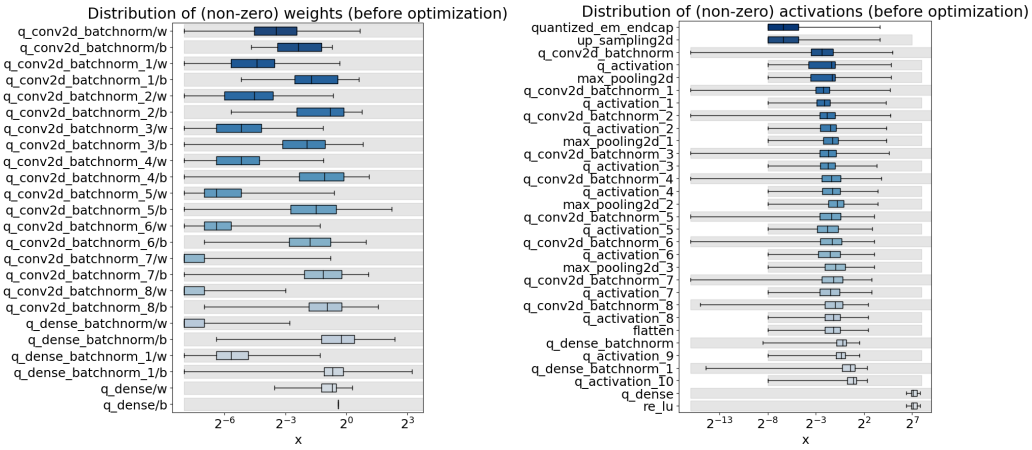
$$-2^{BW_{acc}-1} \leq -2^{BW_{total}-1} \cdot N_{acc} \leq FP_{q,N_{acc}} \leq 2^{BW_{total}-1} \cdot N_{acc} \leq 2^{BW_{acc}-1} - 1 \tag{3}$$

$$BW_{acc} \geq BW_{total} + log_2(N_{acc}) \tag{4}$$

The possible value of FP_q is limited due to the clip function shown in Equation (2). The upper bound and the lower bound of FP_q are amplified by N_{acc} after N_{acc} additions. The inequality of Equation (2) can be further simplified to Equation 4. Obviously, an insufficient bit-width for the accumulator will not satisfy the inequality, which sometimes will cause overflow which cannot be predicted in advance. A deeper model or more filters in layers would cause more additions in one accumulation and thus require wider bit-width to prevent overflow. This results in a proportional increase in the required number of additions, which, in turn, increases the number of bits needed in an accumulator. If an accumulator uses enough bits, overflow could be avoided during computation.

In the original hls4ml, the quantized weights and activation are loaded from the quantized layer in QKeras. The data type is changed from floating-point to fixed-point, which does not cause any quantization error because quantizers have been inserted. On the other hand, in the original hls4ml, the precision of the accumulator will be set the same as the overall model precision. However, as we mentioned in the previous paragraph, the precision of the accumulator is determined by different bit-width according to how many input channels or units are in the layer. As a result, the precision of the accumulator should be adjusted according to the arithmetic characteristics and precision requirement of a layer.

4.1.2 Profiling the HLS Model. It is worth noting that BW_{acc} can be long when the model is large. Therefore, hls4ml provides a tool to reduce the cost of the accumulator by profiling the arithmetic characteristics of the model. By feeding



(a) value range of weight

(b) value range of activation

Fig. 9. The distribution of weights and activation and the range which HLS can represent. (a) shows the value range of weight, and (b) shows the value range of activation. The box plot represents the distribution of value in QKeras, and the gray area is the range which HLS can represent in fixed point data type.

Table 2. The resource utilization of image-only model using different rounding strategies. The detailed FPGA experiment setup is mentioned in Section 5.

Resource	AP_RND	AP_RND_CONV	AP_TRN
BRAM	680	680	680
DSP	3,512	3,516	3,516
LUT	290,972	290,194	289,078
FF	281,209	281,716	280,677
URAM	115	115	115

test data to both the QKeras model and the HLS model in trace mode, the arithmetic computation of all HLS layers in the HLS model can be observed and the BW_{acc} can be further reduced, generally with an acceptable impact on accuracy.

Figure 9a and Figure 9b show the value ranges of weights and activations, respectively. The box plot represents the value in QKeras, and the gray area represents the range of precision that can be represented in HLS. One thing to note is that hls4ml always isolates the activation from the convolution layer and the dense layer, so we can observe the output of the layer and the output of the activation after that layer. Normally, the precision of the output and the accumulator are the same, so we can adjust the bit-width of the accumulator by adjusting the bit-width of the output manually. In other words, we need to make sure that the box plot is within the gray area. Testing the HLS model by providing multiple test data to trace the output of each layer is feasible. However, this process applies only to specific test data and not to all samples or real-world data. Therefore, even if achieving a low bit-width is successful for certain test data, there is still a possibility of incorrect predictions for other test data.

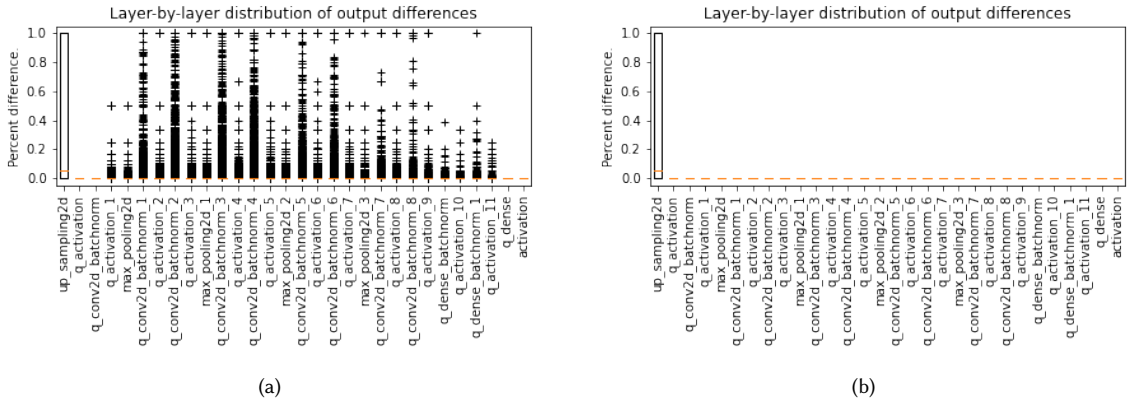


Fig. 10. The distribution of differences in each layer with AP_RND and AP_RND_CONV rounding. (a) shows AP_RND rounding and (b) shows AP_RND_CONV rounding. The black cross represents the outlier of the box plot, and the orange line represents the median of the box plot.

4.2 Other Explorations

4.2.1 Different Rounding Strategies. Due to the difference in rounding between HLS and QKeras, there can be slight variations in the results at the activation layer. The rounding modes used in HLS are discussed in Ref. [50]. Even with a higher number of fractional bits, these differences can propagate to the output of a model. When using QKeras, the *tf.math.round* function rounds values to the nearest even number [51]. However, different data types will adopt different round modes. For example, AP_RND, AP_TRN, or AP_RND_CONV will use rounding to positive infinity, truncating the value, or rounding to the nearest number but the least significant bit must be 0, respectively. The difference in rounding can result in significant variations in predictions, which is shown in Figure 10. Despite this, the increase in resources required when using AP_TRN versus AP_RND_CONV is minimal, as shown in Table 2.

4.2.2 Impact of Non-quantized Input Data. When using QKeras, the first layer will use non-quantized inputs, and perform the computation with quantized weights. The computation on the mixed types will result in a nonquantized output. However, in HLS, all inputs are quantized in the testbench by converting them from floating-point to fixed-point. To generate the quantized QKeras model, there are two solutions: (1) quantize the input outside of the QKeras model and perform Quantization Aware Training (QAT), or (2) insert a linear quantizer directly into the input layer. While the first solution does not require additional hardware overhead, it must be processed in advance. Therefore, in our paper, we adopt the second solution.

4.2.3 Verifying the Quantization Results. For reducing quantization errors or optimizations, we need to ensure that the functionality does not change either from the quantized layer to the HLS layer or from the original layer to the optimized layer. One of the effective ways to verify this is to compare the predictions between the QKeras model and the HLS model. Figures 11a and 11b plot the architecture and precision in each route and can be very helpful in validating the results and debugging the quantization issues. With this enhancement in hls4ml, we can easily identify the difference between the models and quantization errors in various bit-width of weights and activations. This enhancement can also enable fast performance evaluation of the QKeras model using processors (e.g. GPU) instead of waiting for the slow software emulation of the HLS model.

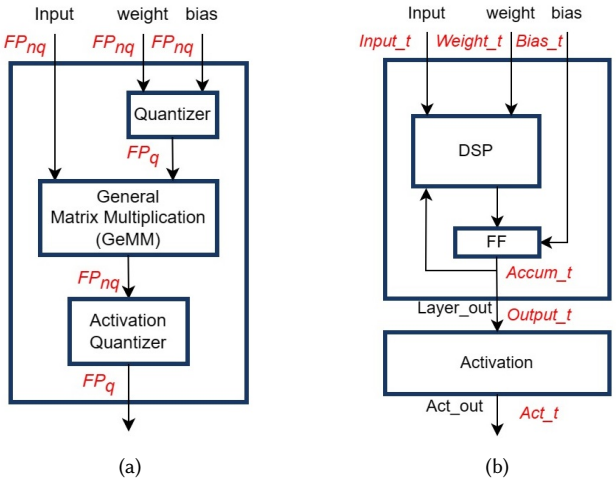


Fig. 11. The architecture of the dense layer in (a) QKeras and (b) HLS model.

5 EXPERIMENTAL RESULTS

This section presents comprehensive experiments with the DeepCalo design and optimizations for FPGAs. The latencies of the models on FPGAs are shown in section 5.1, alongside other performance metrics, and further compared with other computing platforms in Section 5.2. In Section 5.3, we evaluated the performance of two DeepCalo models under different quantization schemes and fixed-point precisions. The resource utilization on FPGAs will be discussed in Section 5.4.

All quantization processes include weights and activations with homogeneous bit width. The fixed-point format is based on the Vivado HLS *ap_fixed* type [52]. Furthermore, the batch normalization folding technique [53] [54] is adopted during the HLS conversion to further lower resource utilization and latency. Specifically, the batch normalization layers that follow the convolutional and dense layers are fused together.

The design is converted to HLS C++ using hls4ml 0.5.1 and then synthesized with Vivado HLS 2019.2, targeting a Xilinx Alveo U50 FPGA with a clock frequency of 200MHz.

5.1 Latency of DeepCalo Models on FPGA

Table 3 presents the latency measurements, expressed in microseconds and cycles, for each layer in both the image-only model and the full model. Latency is defined as the time required to generate a single batch of predictions, and the timing information is derived from the HLS C-synthesis report. To compare the impact of different stream types introduced in Subsection 3.2, a consistent methodology was employed for both models. Initially, all weights and activations were quantized to a precision of <8,2>. Subsequently, during the conversion to HLS, two sets of results were obtained. The first set involved the inclusion of the switch function, as discussed in Subsection 3.3, resulting in a **mixed-type** (partial array-of-streams) dataflow scheme. The second set excluded the switch function, resulting in an **all-single-stream** dataflow scheme.

By incorporating a threshold in the switch function, the array-of-streams approach enabled higher parallelism in the early stage of the CNN, while the remaining layers utilized a single-stream configuration, ensuring data transfer in a

Table 3. Latency comparison for each layer in the (a) image-only model and (b) full model using all-single-stream implementation and mixed-type implementation. Activation layers are negligent. Single-stream is denoted as SS, and array-of-streams is denoted as AS.

(a) CNN (Image-only model)

stream type	**all-single-stream**			**mixed-type**		
Layer	Max Latency			Max Latency		
	µs	cycles	type	µs	cycles	type
Input	-	-	-	-	-	-
Upsampling2D	61.61	12,322	SS	15.41	3,082	AS
Conv2D1	549	109,741	SS	230	46,021	AS
MaxPool1	289	57,793	SS	15.685	3,137	AS
Conv2D2	335	66,991	SS	139	27,841	AS
Conv2D3	405	80,911	SS	139	27,841	AS
MaxPool2	131	26,293	SS	3.995	799	AS
Conv2D4	236	47,281	SS	200	40,081	AS
Conv2D5	275	54,961	SS	275	54,961	SS
MaxPool3	59.785	11,957	SS	59.785	11,957	SS
Conv2D6	175	34,921	SS	175	34,921	SS
Conv2D7	310	62,065	SS	310	62,065	SS
MaxPool4	27.225	5,445	SS	27.225	5,445	SS
Conv2D8	194	38,801	SS	194	38,801	SS
Conv2D9	212	42,376	SS	212	42,376	SS
Dense1	35.885	7,177	SS	35.885	7,177	SS
Dense2	5.165	1,033	SS	5.165	1,033	SS
Dense3	1.305	261	SS	1.305	261	SS
Total	802	160,437	-	343	68,531	-

(b) Full model

stream type	**all-single-stream**			**mixed-type**		
Layer	Max Latency			Max Latency		
	µs	cycles	type	µs	cycles	type
Scalar Net						
Input Scalar Variables	-	-	-	-	-	-
Dense	1.555	311	SS	1.555	311	SS
Track Net						
Input Track Vectors	-	-	-	-	-	-
Time Distributed	199	39,756	SS	199	39,756	SS
Sum1D	49.96	9,992	SS	49.96	9,992	SS
Dense1	2.61	522	SS	2.61	522	SS
Dense2	2.61	522	SS	2.61	522	SS
FiLM Generator (Connecting the Track and Scalar Net)						
Concatenate	3.87	774	SS	3.87	774	SS
Dense1	8.375	1,675	SS	8.375	1,675	SS
Dense2	12.855	2,571	SS	12.855	2,571	SS
Dense3	20.375	4,075	SS	20.375	4,075	SS
FiLM1	247	49,318	SS	15.59	3,118	AS
FiLM2	121	24,262	SS	4.13	826	AS
FiLM3	58.91	11,782	SS	58.91	11,782	SS
FiLM4	28.19	5,638	SS	28.19	5,638	SS
FiLM5	14.11	2,822	SS	14.11	2,822	SS
Combined with all-single-stream CNN	873	174,600	-	-	-	-
Combined with mixed-type CNN	-	-	-	374	74,734	-

pipelined manner with low resource demand. As the full model does not have a sequential structure, the additional submodels had minimal impact on the overall latency result. Comparatively, the array-of-streams approach exhibited significantly shorter latency in the layers, leading to reduced overall latency. Both models achieved a speed improvement of approximately 2.34×.

Notably, in the all-single-stream implementation, the first convolutional layer experienced longer latency compared to the other layers. This increase in latency is primarily attributed to the larger dimension of the input feature map and the inherent complexity of convolutions, which ultimately creates a bottleneck within the pipeline.

5.2 Comparisons with Other Computing Platforms

In this subsection, we present a comparison of the image-only model and the full model on various processing platforms, including CPUs, GPUs, and FPGAs. The comparison focuses on latency, speedup, power consumption, and energy consumption. Speedup is calculated by normalizing the latency of the Ryzen 5 5600H CPU. The experiments were carried out using three different batch settings: 1, 5, and 100. Note that a batch size of 5 corresponds to approximately

Table 4. Performance comparisons of the (a) image-only model and (b)full model on processing platforms: CPUs, GPUs, and FPGAs. Both models have floating-point precision for the CPUs and GPUs, whereas the precision for the FPGA is ap_fixed<8,2>.

(a) CNN (Image-only model)

Coprocessor	CPU			GPU			FPGA	
Type	Ryzen 7 3700X	Ryzen 5 5600H	Intel i5-12400F	RTX 2070 Super	Tesla V100	RTX 2080 Ti	single-stream	mixed-type
Batch=1								
Latency	6ms	6.227ms	4.439ms	5.98ms	3.5ms	5.8ms	0.697ms	0.443ms
Speedup	1.038×	**1**×	1.403×	1.041×	1.779×	1.074×	8.934×	**14.056**×
Power	53.82W	27.38W	40.38W	35.68W	61.08W	67.83W	18.33W	20W
Energy	322.92mJ	170.495mJ	179.247mJ	213.366mJ	213.78mJ	393.414mJ	12.778mJ	8.86mJ
Batch=5								
Latency	10ms	10.165ms	8.03ms	8ms	3.6ms	6ms	2.395ms	1.34ms
Speedup	1.017×	**1**×	1.266×	1.271×	2.824×	1.694×	4.244×	7.586×
Power	62.03W	35.22W	47.29W	40.30W	62.31W	66.23W	19W	23W
Energy	620.3mJ	358.01mJ	379.74mJ	322.4mJ	224.315mJ	397.38mJ	45.505mJ	30.82mJ
Batch=100								
Latency	50ms	72.8ms	49.3ms	8.4ms	4.8ms	6.7ms	44.4ms	23.5ms
Speedup	1.456×	**1**×	1.477×	8.667×	15.167×	10.866×	1.64×	3.098×
Power	81.02W	40.54W	62.46W	93.25W	92.22W	87.21W	19W	24W
Energy	4.051J	2.951J	3.079J	0.783J	0.443J	0.584J	0.844J	0.564J

(b) Full model

Coprocessor	CPU			GPU			FPGA	
Type	Ryzen 7 3700X	Ryzen 5 5600H	AMD EPYC 7262	RTX 2070 Super	Tesla V100	RTX 2080 Ti	single-stream	mixed-type
Batch=1								
Latency	7.52ms	8.75ms	5.865ms	8.47ms	4.8ms	8.2ms	1.106ms	0.898ms
Speedup	1.164×	**1**×	1.492×	1.033×	1.823×	1.067×	7.911×	**9.744**×
Power	53.73W	29.13W	42.65W	49.77W	60.11W	64.54W	19.76W	20.75W
Energy	404.05mJ	254.888mJ	250.142mJ	421.552mJ	288.528mJ	529.228mJ	21.855mJ	18.634mJ
Batch=5								
Latency	11.5ms	13.45ms	10.545ms	9.75ms	5.1ms	7ms	2.695ms	1.485ms
Speedup	1.17×	**1**×	1.275×	1.379×	2.637×	1.921×	4.991×	9.057×
Power	62.44W	37.67W	48.94W	51.83W	61.73W	84.18W	21W	23.775W
Energy	718.06mJ	506.66mJ	516.07mJ	505.345mJ	314.825mJ	589.26mJ	56.595mJ	35.305mJ
Batch=100								
Latency	86ms	119ms	91.4ms	15ms	7.1ms	9.5ms	53.9ms	29.7ms
Speedup	1.384×	**1**×	1.302×	7.933×	16.761×	12.526×	2.208×	4.007×
Power	86.51W	47.24W	75.41W	93.59W	116.4W	112.74W	21W	23.833W
Energy	7.44J	5.622J	6.893J	1.404J	0.826J	1.071J	1.132J	0.708J

the number of electron collisions in LHC scenarios. Compared to the SONIC framework mentioned in subsection 2.4, our implementation has shown a speedup of 5.6× with a batch size of 5.

5.2.1 Measurement Methodologies. To ensure unbiased results, we conducted 10^5 repetitions for each measurement of latency, power consumption, and energy consumption on CPUs and GPUs. The average values were then calculated from these repetitions.

The power consumption of the GPUs was measured using Nvidia's built-in command, which recorded the power readings from specific registers on the GPUs at intervals of 10 milliseconds. On the other hand, the power consumption of CPUs was measured by sampling the CPU package power every 20 milliseconds.

In the case of FPGA, we examined the two dataflow schemes: mixed-type and all-single-stream. Latency was assessed by measuring the time it took to load the input data from the DRAM, perform computations in the processing engines,

and write the results back to the DRAM. To mitigate the influence of dynamic factors during run-time, we obtained the average latency through 10^4 runs. FPGA power consumption was measured as the average of 200 runs, using the "query" command within the Xilinx Runtime Library (XRT) [55].

5.2.2 Image-only Model Analysis. The results of the image-only model are presented in Table 4a. As the batch size increases for both CPUs and GPUs, the latency per batch tends to decrease, while the power consumption increases. GPUs demonstrate significantly improved performance when the batch size is increased to 100, owing to their superior parallelism compared to CPUs.

In the case of FPGA designs, the mixed-type scheme exhibits notably shorter latency across all batch sizes compared to the all-single-stream scheme. It achieves a speedup of 1.573× for batch size 1, 1.787× for batch size 5, and 1.889× for batch size 100. Due to the improved pipelined dataflow mentioned in Section 3, both dataflow schemes demonstrate decreasing latency per batch with increasing batch size. This efficiency also leads to a slightly higher power consumption as the batch size increases.

For batch sizes of 1 and 5, the speedup ratio and energy consumption of FPGA designs surpass those of CPUs and GPUs. However, for batch size 100, GPUs outperform FPGAs in terms of speedup due to the massively parallel processing capabilities. Nevertheless, FPGAs still exhibit significantly better speedup compared to CPUs. Regarding energy consumption, the performance of the FPGA is slightly inferior to that of the GPU-Tesla V100. However, it is important to consider that the Alveo U50 FPGA costs approximately US$3,000 [56] while the Tesla V100 GPU is priced at around US$15,000 [57].

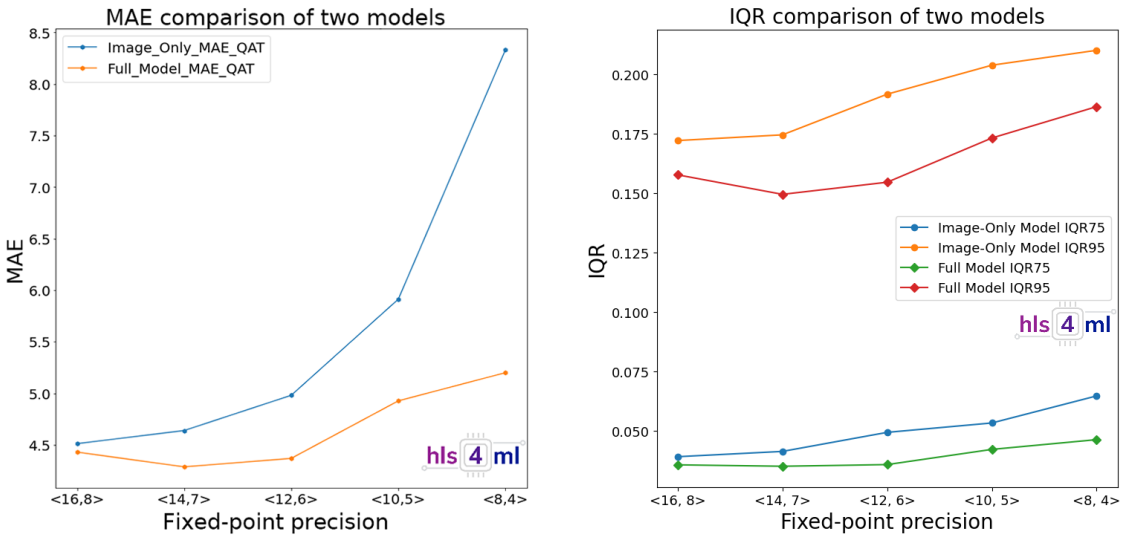
5.2.3 Full Model Analysis. The performance results for the full model are listed in Table 4b. The mixed-type scheme attains 1.231×, 1.815×, and 1.815× faster than the all-single-stream scheme for batch sizes 1, 5 and 100, respectively. Due to the inclusion of additional side branches from the submodels (Track-Net and Scalar-Net), traffic jams are more likely to occur in certain parts of the FIFOs within the full model. Therefore, the efficiency of the dataflow pipeline is reduced, leading processing units to reach their maximum capacity when the batch size becomes excessively large.

Similarly to the observations in Table 4a, the latency per batch decreases, while the power consumption slightly increases as the batch size increases. Compared to CPUs and GPUs, FPGA designs still exhibit lower latency and energy consumption for batch sizes 1 and 5. Although GPUs outperform FPGAs in terms of speedup for batch size 100, the FPGA designs continue to demonstrate lower energy consumption compared to GPUs.

5.3 Evaluation of Performance

This section examines in detail the effects of using the QAT and PTQ approaches. We start by providing hyperparameters used in the QAT procedure. Next, we define the two evaluation metrics to assess model performance: mean absolute error (MAE) and interquartile range (IQR). MAE provides a measure of the average prediction error, while IQR captures the distribution tendencies. Finally, we compare the performance of the image-only model and the full model using the defined evaluation metrics in both quantization schemes. The goal of the evaluation process is to identify the optimal quantization scheme that balances performance and resource utilization, which is discussed in the next subsection.

5.3.1 Quantization-aware Training Hyperparameters. We use the same set of hyperparameters as those used for training floating-point models. The dataset contains about 1.2 million electrons from the ECAL endcap region for the energy regression task. It was divided approximately 70%, 15%, and 15% into training, validation, and test set. Both models are trained with the *Nadam* optimizer [58] alone with a *cyclical learning rate* (CLR) [59] schedule for 300 epochs. For



(a) Evaluation metric: MAE

(b) Evaluation metrics: IQR_{75} and IQR_{95}

Fig. 12. Performance comparison of the image-only model and full model as a function of model precision after QAT. Evaluation metrics are (a) MAE and (b) IQR. Both models' total bits vary from 16 to 8, with the integer bits set to half of the total bits.

CLR, the stepsize is 3 epochs, the baseline learning rate is $5e^{-4}$ and the maximum learning rate is $7e^{-2}$. The initial learning rate is determined by scanning between $1e^{-5}$ and $1e^{-2}$ for an epoch before training. The batch size is set to 1,024, and we employ the *logcosh* loss function to reduce sensitivity to outliers. An early stopping with *min_delta* of 0.001 and *patience* of 150 monitored on the validation loss is adopted. The details of the hyperparameter search and dataset collection are discussed in Ref. [12].

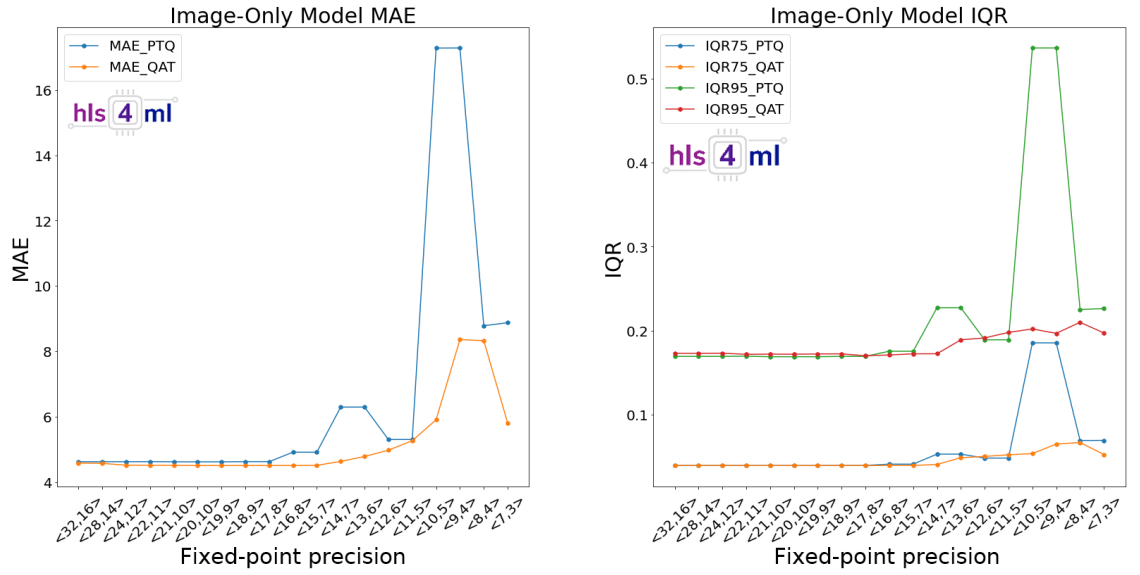
5.3.2 Evaluation Metrics. The MAE metric is defined in Equation 5, where $\hat{y}$ is the predicted energy and y is the true energy, and the subscript i denotes the i^{th} data in the total number of data points n.

The IQR of the distribution of relative errors (RE) metric is also used to provide a more robust overview, which is defined in Equation 6. It helps to identify the range within which a certain percentage of REs fall, with P_m representing the m^{th} percentile. In this paper, we utilized IQR_{75} and IQR_{95}.

$$MAE = \frac{1}{n}\sum_{i=1}^{n}|\hat{y}_i - y_i| \tag{5}$$

$$IQR_m(RE) = P_m(RE) - P_{100-m}(RE), \quad RE = \frac{\hat{y} - y}{y} \tag{6}$$

5.3.3 Quantization Schemes Performance Comparison. Balancing model performance and resource utilization is a complex and time-consuming task when optimizing QAT configuration. To tackle this challenge, we adopt a two-step approach. We first conducted an extensive search for the bit widths of both models using PTQ, which serves as a reference baseline. For the image-only model, it ranges from 32 total bits down to 2 total bits, while also varying the number of integer bits. However, due to the increased complexity of the full model, we only went through this process



(a) Image-only model MAE

(b) Image-only model IQR

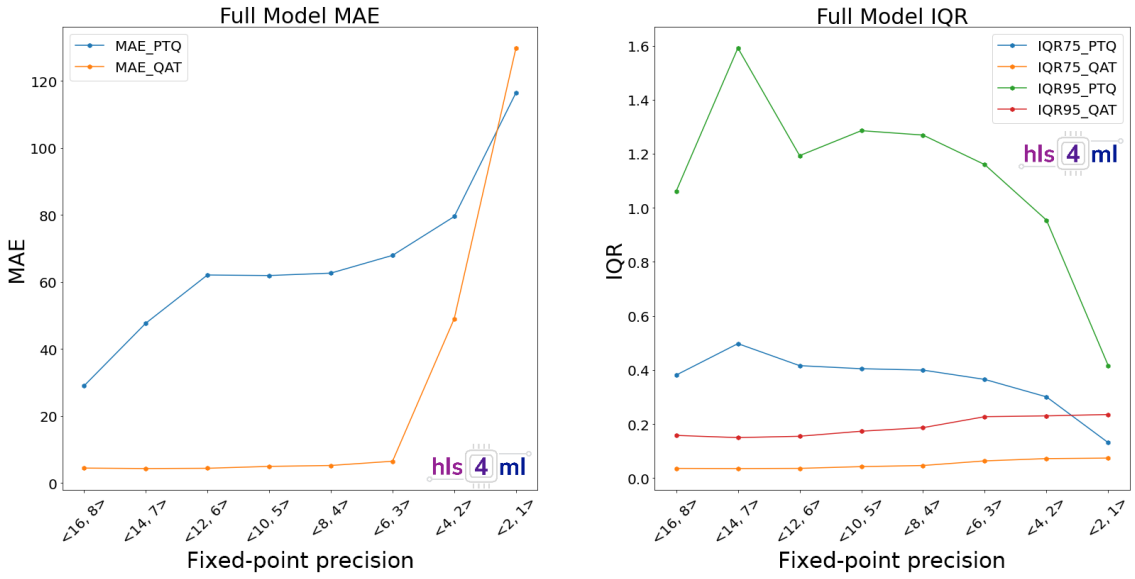
Fig. 13. Performance of the image-only model as a function of model precision using PTQ and QAT. Evaluation metrics are (a) MAE and (b) IQR. The total bits ranged from 32 to 7, where the integer bits were set to half of the total bits.

from 16 total bits to 2 total bits. During the PTQ evaluation, we observed that providing similar bits for both the integer and fraction parts yielded the lowest MAE for both models. This finding guided our subsequent QAT experiments, where we set integer bits as half of the total bits.

Figure 12 presents two performance comparisons between the image-only model and the full model after performing QAT with different model precisions, using the two evaluation metrics mentioned above. Concerning both metrics, the full model exhibits better regression performance and robustness compared to the image-only model, regardless of precision. This finding suggests that the inclusion of additional components and features in the full model enhances its performance, even with various precision settings.

Figure 13 includes two subfigures evaluating the impact of the PTQ and QAT schemes on the performance of the image-only model. For Figure 13a, the MAEs of the QAT and PTQ models remain relatively consistent, from 32 to 17 total bits. However, as the total bit width is further reduced, there is a gradual decline in the performance of both models. However, the MAE of the QAT model is always lower than that of the PTQ model. We did not include bits lower than 7 since the performance has dropped to an unreliable level. In Figure 13b, the IQR distribution pattern mirrors the MAE trend we observe in Figure 13a. Furthermore, there are sudden peaks of MAE and IQR in the PTQ model around 10 total bits. This suggests that 10 total bits are not sufficient to accurately represent the weight values in the PTQ model.

Figure 14 presents two subfigures evaluating the impact of the PTQ and QAT schemes on the performance of the full model. In Figure 14a, the MAE remains consistently low as the total bits range from 16 bits to 3 bits using QAT. On the contrary, the PTQ model starts with a relatively higher MAE and exhibits an increasing trend as the total bits decrease. Moreover, the PTQ model generally has a higher MAE compared to the QAT model, with the gaps narrowing only when the total bits are extremely low. Similarly, Figure 14b reveals that the IQR of the PTQ model is higher than that of



(a) Full model MAE (b) Full model IQR

Fig. 14. Performance of the full model as a function of model precision using PTQ and QAT. Evaluation metrics are (a) MAE and (b) IQR. The total bits ranged from 16 to 2, where the integer bits were set to half of the total bits.

the QAT model for most precision levels. Remarkably, the QAT model demonstrates a feature of consistently low values of IQR_{75} and IQR_{95} in various precisions, which underlines its robustness to changes in precision levels.

5.3.4 Analysis of Model Performance. Based on the observations in Figure13 and Figure14, it suggests that enough bits for both integers and fractions are essential for such a regression task. As the bit width decreases, the model's predictions become more varied and extreme, suggesting the significance of bit width in regulating the prediction dependability.

Furthermore, when different quantization strategies are compared, QAT consistently exhibits superior performance over PTQ at nearly every precision level. This underscores the robustness and efficacy of QAT in preserving model performance while reducing resource utilization.

5.4 Resource Utilization

5.4.1 Stream Types Resource Comparison. Table 5 presents the resource consumption and latency of two convolutional layers extracted from the image-only model, one with 16 filters, an input channel size of 4, and a precision of <32,16>, and the other with 256 filters, an input channel size of 256, and a precision of <16,8>. Both layers were evaluated using different stream types.

In Table 5a, the resource consumption and latency in array-of-streams are quite similar to those in stream-of-struct. This indicates that when the number of channels is small, stream-of-struct can be efficiently processed using Vivado HLS. For BRAM usage, it is used for both read-only memory (ROM) and FIFOs purposes. ROM is utilized for the storage of weight data, so the value is the same for all three data types. FIFOs are used for data transmission. In this experiment,

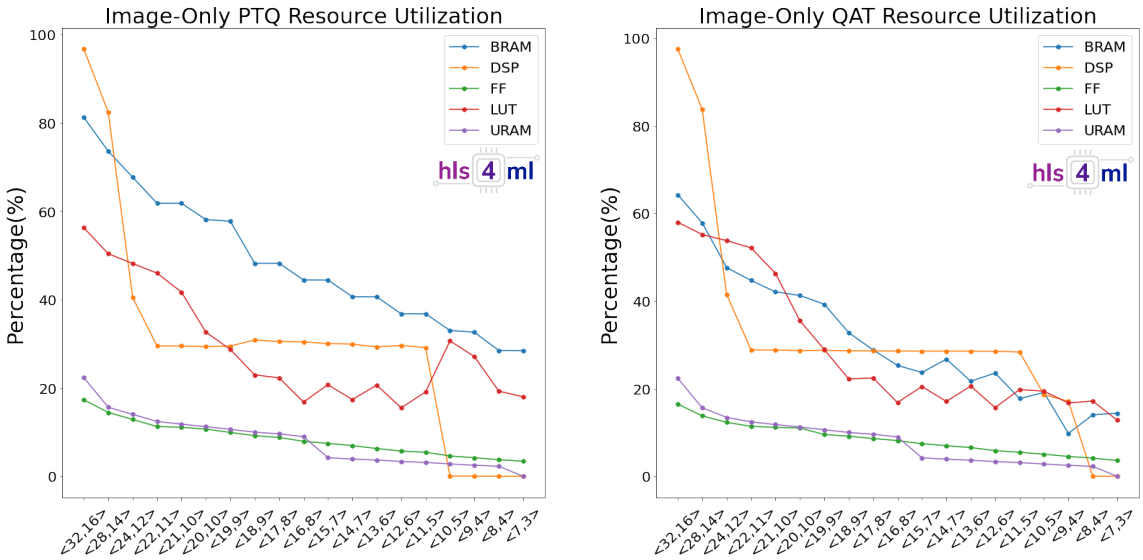
Table 5. Resources utilization and latencies of two convolutional layers with different stream types.

(a) 16 filters with precision of <32,16>

	single	array	struct
BRAM (ROM/FIFO)	36 (28/8)	60 (28/32)	60 (28/32)
DSP	126	126	126
FF	10,123	9,761	9,758
LUT	26,623	27,396	27,382
Cycles	230,224	176,943	173,863

(b) 256 filters with precision of <16,8>

	single	array	struct
BRAM (ROM/FIFO)	912 (911/1)	1167 (911/256)	1167 (911/256)
DSP	513	513	513
FF	82,520	102,956	129,419
LUT	69,143	69,574	106,643
Cycles	19,422	10,775	20,526



(a) Image-only model PTQ resource utilitzation

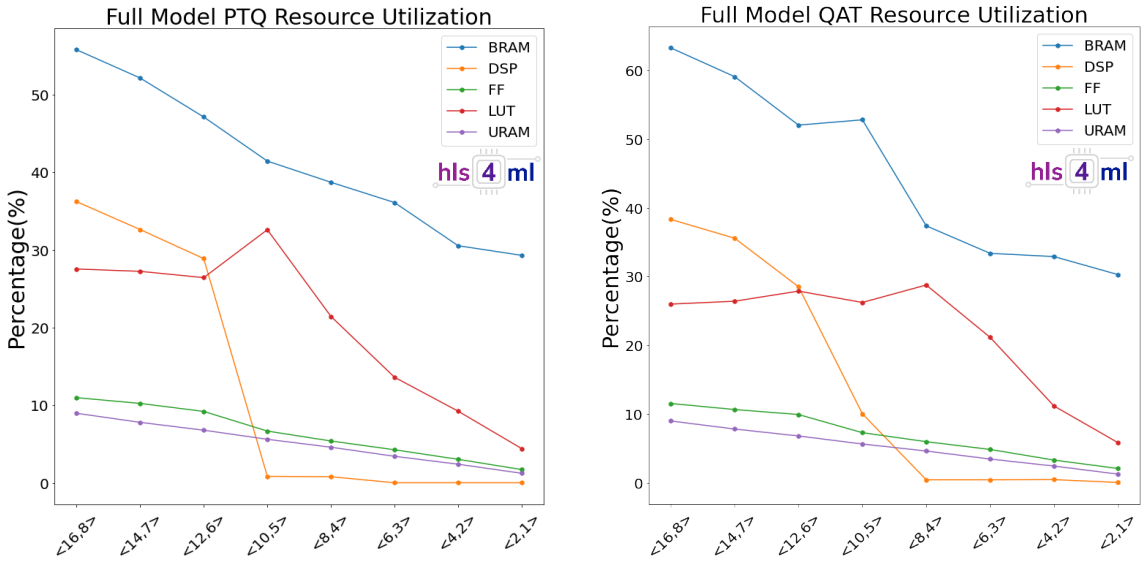
(b) Image-only model QAT resource utilitzation

Fig. 15. Resource consumption analysis of the image-only model using different quantization schemes. The target FPGA is Alveo U250.

one FIFO accounts for 8 BRAMs. Single-stream consumes one FIFO, while both array-of-streams and stream-of-struct consume 4 FIFOs, which is equal to the input channel size for this 16-filter convolutional layer.

In Table 5b, the latency of stream-of-struct is larger than that of single-stream, and the resource utilization is the highest in this table. In BRAM usage, the ROM is also used for the storage of weight data, and the value is the same in different data types. In this experiment, 1 FIFO accounts for 1 BRAM due to the difference in precision compared to Table 5a. There are 256 input channels for this 256-filter convolutional layer, so both array-of-streams and stream-of-struct consume 256 FIFOs.

5.4.2 Deepcalo Models Resource Comparison. Figure 15 and Figure 16 show the resource consumption of the image-only model and full model respectively, in relation to the total available resources. The target FPGA is Alveo U250 [60],



(a) Full model PTQ resource utilitzation

(b) Full model QAT resource utilitzation

Fig. 16. Resource consumption analysis of the full model using different quantization schemes. The target FPGA is Alveo U250.

currently the largest platform available in the Xilinx Alveo FPGA series, to ensure that the available resources are sufficient at a high bit width. UltraRAM (URAM), a storage unit similar to BRAM, is utilized as buffer in the convolutional layers of our design. There are five curves representing different resources, all of which exhibit a downward trend as the total bit width decreases.

In Figure 15a, the resource consumption of the image-only model with PTQ is presented. The limitation for FPGA inference in high bitwidth is DSPs, and this phenomenon is also evident in Figure 15b. The DSP consumption decreases significantly from 32 bits to 22 bits, followed by a stable trend until 11 bits. This is due to the maximum input size for multiplication in DSP48E2 being 27×18 [61]. If an input exceeds this limit, two DSPs will be applied to perform the multiplication. For bit widths below 11 bits, the DSP utilization decreases to nearly zero percent since the multiplication is carried out by LUTs, resulting in an increase in LUT consumption from 11 bits to 10 bits. The second limiting resource is BRAM. The image-only model containing 1.8M parameters is quite large, and BRAM usage accounts for approximately 80% in a bit width of 32. Therefore, to maintain desirable performance while reducing BRAM consumption and computing resources, it is essential to apply QAT.

In Figure 15b, the resource consumption of image-only model with QAT is depicted. The curve for DSP consumption is very similar to that shown in Figure 15a. For bit widths below 11 bits, the DSP consumption decreases smoothly until it reaches 8 bits, which is in contrast to the abrupt trend observed in PTQ. The reason for this discrepancy is that the training method used for QAT is different from that one for PTQ, resulting in different precision levels for accumulators and output data in Conv2D and Dense layer. What is even more interesting is that the curve for BRAM consumption decreases 20% compared to that shown in Figure15a. The is because the weight data in BRAM can be significantly optimized in logic synthesis.

In Figure 16a, the resource consumption of the full model with PTQ is illustrated. In the curve for DSP consumption, the transition point also occurs between 12 bits and 10 bits, and as a result, the LUT consumption increases. The resource consumption of full model with QAT is presented in Figure 16b. In the curve for DSP consumption, also due to different training method used for PTQ and QAT, the trend decreases gradually between a bit width of 12 and 8. Different training method also has an impact on the BRAM usage. Unlike the result in image-only model, full model utilizing PTQ and QAT exhibits very similar BRAM usage. As FIFO is established by BRAM, a higher precision level used for output data will lead to increased BRAM consumption. In full model with QAT, output precision of certain layers is double to that of the corresponding layers in PTQ, which leads to higher consumption of BRAM in FIFOs. Therefore, the BRAM consumption in QAT is more than that one in PTQ. Overall, the trend in QAT for different resources is very similar to that shown in PTQ.

6 CONCLUSION

In this paper, we present an automated design and optimization workflow based on hls4ml to deploy DeepCalo models on a Xilinx Alveo U50 FPGA, with potential applicability to other large CNNs. We highlight the importance of choosing the appropriate stream-based dataflow to balance resource utilization and achieve the desired latency. To ensure the accuracy and reliability of the converted HLS models, we illustrate our approach to eliminating potential quantization errors at an early stage, prior to conversion to HLS. To validate our methodology, we compare the FPGA implementation of both models with other co-processors using key performance indicators such as latency, speedup, power consumption, and energy efficiency. Examining actual LHC conditions, the image-only model yielded an inference latency of 1.34 ms for every 5 images, which is 5.6 times faster than the existing GPU-based system. At a batch size of one, the image-only model demonstrates a latency of 0.443 ms, while the full model exhibits a latency of 1.34 ms, achieving the Level-1 Trigger requirement. Compared to the Ryzen-5600H CPU and the Tesla V100 GPU, the image-only (full) model achieves speedups of up to 14.1× (9.7×) and 7.9× (5.3×), respectively. Finally, we show that quantization-aware training significantly reduces resource usage and preserves performance compared to post-training quantization.

ACKNOWLEDGMENTS

We would like to express our sincere appreciation to the Fast Machine Learning Collaboration for their invaluable contributions and stimulating discussions that greatly influenced this research. Our heartfelt gratitude goes to Frederik Faye for his exceptional vision and dedication in initiating and leading the DeepCalo project, which laid the groundwork for our study. Additionally, we extend our thanks to the Accelerated AI Algorithms for Data-Driven Discovery (A3D3) Institute for valuable support.

CODE AVAILABILITY STATEMENT

The source code used in this paper has been organized and made available at https://github.com/ChiRuiChen/ACFPER, which includes the code for the implementation of DeepCalo, HLS4ML, and FPGA.

REFERENCES

[1] Particle Data Group et al. 2008. Review of particle physics. *Physics Letters B*, 667, 1-5, (Sept. 2008), 1–6. A booklet is available containing the Summary Tables and abbreviated versions of some of the other sections of this full Review. All tables, listings, and reviews (and errata) are also available on the Particle Data Group website: http://pdg.lbl.gov. DOI: 10.1016/j.physletb.2008.07.018.

[2] Lyndon Evans and Philip Bryant. 2008. Lhc machine. *Journal of Instrumentation*, 3, 08, (Aug. 2008), S08001. DOI: 10.1088/1748-0221/3/08/S08001.

[3] 2016. Cern accelerating science. (n.d.) https://home.cern/science/accelerators/large-hadron-collider.

[4] ATLAS Collaboration. 2012. Observation of a new particle in the search for the standard model higgs boson with the atlas detector at the lhc. *Physics Letters B*, 716, 1, 1–29. DOI: https://doi.org/10.1016/j.physletb.2012.08.020.

[5] Dan Guest, Kyle Cranmer, and Daniel Whiteson. 2018. Deep learning and its application to lhc physics. *Annual Review of Nuclear and Particle Science*, 68, 1, 161–181. eprint: https://doi.org/10.1146/annurev-nucl-101917-021019. DOI: 10.1146/annurev-nucl-101917-021019.

[6] Frederik G. Faye. 2019. Deepcalo. https://gitlab.com/ffaye/deepcalo.

[7] F. Chollet et al. 2015. Keras. https://https://keras.io/.

[8] ATLAS Collaboration. 2010. The ATLAS simulation infrastructure. *The European Physical Journal C*, 70, 3, (Sept. 2010), 823–874. DOI: 10.1140/epjc/s10052-010-1429-9.

[9] ATLAS collaboration. 2017. Electron and photon reconstruction and performance in ATLAS using a dynamical, topological cell clustering-based approach. Tech. rep. All figures including auxiliary figures are available at https://atlas.web.cern.ch/Atlas/GROUPS/PHYSICS/PUBNOTES/ATL-PHYS-PUB-2017-022. CERN, Geneva. https://cds.cern.ch/record/2298955.

[10] ATLAS collaboration. 2019. Electron and photon performance measurements with the ATLAS detector using the 2015–2017 LHC proton-proton collision data. *Journal of Instrumentation*, 14, 12, (Dec. 2019), P12006–P12006. DOI: 10.1088/1748-0221/14/12/p12006.

[11] Darin Acosta et al. 2018. Boosted decision trees in the level-1 muon endcap trigger at cms. *Journal of Physics: Conference Series*, 1085, 4, (Sept. 2018), 042042. DOI: 10.1088/1742-6596/1085/4/042042.

[12] 2019. Energy reconstruction of electrons and photons using convolutional neural networks, master's thesis (cand.scient.) https://gitlab.com/ffaye/deepcalo/-/blob/master/demos/atlas_specific_usecases/train_recommended_models/thesis.pdf.

[13] Pallabi Das and on behalf of the CMS Collaboration. 2022. An overview of the trigger system at the cms experiment. *Physica Scripta*, 97, 5, (Apr. 2022), 054008. DOI: 10.1088/1402-4896/ac6302.

[14] ATLAS collaboration. 2020. Operation of the atlas trigger system in run 2. *Journal of Instrumentation*. Query date: 2023-03-15 23:42:14. DOI: 10.1088/1748-0221/15/10/P10004.

[15] The ATLAS TDAQ Collaboration. 2016. The atlas data acquisition and high level trigger system. *Journal of Instrumentation*, 11, 06, (June 2016), P06008. DOI: 10.1088/1748-0221/11/06/P06008.

[16] ATLAS collaboration. 2020. Performance of the cms level-1 trigger in proton-proton collisions at s = 13 tev. *Journal of Instrumentation*, 15, 10, (Oct. 2020), P10017. DOI: 10.1088/1748-0221/15/10/P10017.

[17] Jeffrey Krupa et al. 2021. GPU coprocessors as a service for deep learning inference in high energy physics. *Machine Learning: Science and Technology*, 2, 3, (Apr. 2021), 035005. DOI: 10.1088/2632-2153/abec21.

[18] Burkhard Schmidt. 2016. The high-luminosity upgrade of the lhc: physics and technology challenges for the accelerator and the experiments. *Journal of Physics: Conference Series*, 706, 2, (Apr. 2016), 022002. DOI: 10.1088/1742-6596/706/2/022002.

[19] ATLAS and CMS Collaborations. 2019. Report on the physics at the hl-lhc and perspectives for the he-lhc. (2019). DOI: 10.48550/ARXIV.1902.10229.

[20] Javier Duarte et al. 2018. Fast inference of deep neural networks in FPGAs for particle physics. *JINST*, 13, 07, P07027. arXiv: 1804.06913 [physics.ins-det]. DOI: 10.1088/1748-0221/13/07/P07027.

[21] Jennifer Ngadiuba et al. 2021. Compressing deep neural networks on FPGAs to binary and ternary precision with HLS4ML. *Mach. Learn. Sci. Tech.*, 2, 015001. arXiv: 2003.06308 [cs.LG]. DOI: 10.1088/2632-2153/aba042.

[22] Elham E Khoda et al. 2022. Ultra-low latency recurrent neural network inference on fpgas for physics applications with hls4ml. (2022). DOI: 10.48550/ARXIV.2207.00559.

[23] Thea Aarrestad et al. 2021. Fast convolutional neural networks on FPGAs with hls4ml. *Mach. Learn. Sci. Tech.*, 2, 4, 045015. arXiv: 2101.05108 [cs.LG]. DOI: 10.1088/2632-2153/ac0ea1.

[24] Abdelrahman Elabd et al. 2022. Graph neural networks for charged particle tracking on FPGAs. *Frontiers in Big Data*, 5, (Mar. 2022). arXiv: 2112.02048. DOI: 10.3389/fdata.2022.828666.

[25] S. Summers et al. 2020. Fast inference of boosted decision trees in FPGAs for particle physics. *Journal of Instrumentation*, 15, 05, (May 2020), P05026–P05026. DOI: 10.1088/1748-0221/15/05/p05026.

[26] FastML Team. 2021. Fastmachinelearning/hls4ml. https://github.com/fastmachinelearning/hls4ml.

[27] Claudionor N. Coelho et al. 2021. Automatic heterogeneous quantization of deep neural networks for low-latency inference on the edge for particle detectors. *Nature Machine Intelligence*, 3, 8, (June 2021), 675–686. DOI: 10.1038/s42256-021-00356-5.

[28] Claudionor N. Coelho Jr et al. 2019. Qkeras. https://github.com/google/qkeras.

[29] Xilinx. 2023. Alveo u50 data center accelerator card. https://www.xilinx.com/products/boards-and-kits/alveo/u50.html.

[30] Xilinx. 2023. Amd ryzen 5 5600h. https://www.amd.com/en/products/apu/amd-ryzen-5-5600h.

[31] NVIDIA. 2023. Nvidia v100 tensor core. https://www.nvidia.com/en-us/data-center/v100/.

[32] Michaela Blott, Thomas B Preußer, Nicholas J Fraser, Giulio Gambardella, Kenneth O'brien, Yaman Umuroglu, Miriam Leeser, and Kees Vissers. 2018. Finn-r: an end-to-end deep-learning framework for fast exploration of quantized neural networks. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, 11, 3, 1–23.

[33] Yaman Umuroglu, Nicholas J. Fraser, Giulio Gambardella, Michaela Blott, Philip Leong, Magnus Jahre, and Kees Vissers. 2017. Finn: a framework for fast, scalable binarized neural network inference. FPGA '17, 65–74.

[34] Yu-Hsin Chen, Tushar Krishna, Joel Emer, and Vivienne Sze. 2017. Eyeriss: an energy-efficient reconfigurable accelerator for deep convolutional neural networks. In *IEEE Journal of Solid-State Circuits* number 1. Vol. 52. IEEE, 127–138.

[35] Zhu Qiu, Jason Cong, and Youxiang Li. 2021. Flexcnn: a flexible systolic array-based fpga accelerator for convolutional neural networks. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*.

[36] 2023. Xilinx vitis ai. https://github.com/Xilinx/Vitis-AI.

[37] Nimmy M Philip and N M Sivamangai. 2022. Review of fpga-based accelerators of deep convolutional neural networks. In *2022 6th International Conference on Devices, Circuits and Systems (ICDCS)*, 183–189. DOI: 10.1109/ICDCS54290.2022.9780689.

[38] Yuhao Wu. 2023. Review on fpga-based accelerators in deep learning. In *2023 IEEE 6th Information Technology,Networking,Electronic and Automation Control Conference (ITNEC)*. Vol. 6, 452–456. DOI: 10.1109/ITNEC56291.2023.10082175.

[39] Yunxiang Hu, Yuhao Liu, and Zhuovuan Liu. 2022. A survey on convolutional neural network accelerators: gpu, fpga and asic. In *2022 14th International Conference on Computer Research and Development (ICCRD)*, 100–107. DOI: 10.1109/ICCRD54409.2022.9730377.

[40] Nicolò Ghielmetti et al. 2022. Real-time semantic segmentation on FPGAs for autonomous vehicles with hls4ml. *Mach. Learn. Sci. Tech.* arXiv: 2205.07690 [cs.CV]. DOI: 10.1088/2632-2153/ac9cb5.

[41] ATLAS collaboration. 2018. Measurements of higgs boson properties in the diphoton decay channel with 36 fb^{-1} of pp collision data at $\sqrt{s} = 13$ TeV with the atlas detector. *Phys. Rev. D*, 98, (Sept. 2018), 052005, 5, (Sept. 2018). DOI: 10.1103/PhysRevD.98.052005.

[42] K Binder, D Heermann, L Roelofs, and ... 1993. Monte carlo simulation in statistical physics. *Computers in …* Query date: 2023-03-15 21:17:26. DOI: 10.1063/1.4823159.

[43] Karen Simonyan and Andrew Zisserman. 2015. Very deep convolutional networks for large-scale image recognition. (2015). arXiv: 1409.1556 [cs.CV].

[44] Ethan Perez, Florian Strub, Harm de Vries, Vincent Dumoulin, and Aaron C. Courville. 2017. Film: visual reasoning with a general conditioning layer. *CoRR*, abs/1709.07871. http://arxiv.org/abs/1709.07871 arXiv: 1709.07871.

[45] Claudionor N. Coelho et al. 2021. Automatic heterogeneous quantization of deep neural networks for low-latency inference on the edge for particle detectors. *Nature Machine Intelligence*, 3, 8, (June 2021), 675–686. DOI: 10.1038/s42256-021-00356-5.

[46] K. Pedro. 2020. Soniccms. https://github.com/fastmachinelearning/SonicCMS.

[47] Xilinx. 2019. Vivado hls 2019.1: pragma hls dataflow. https://www.xilinx.com/htmldocs/xilinx2019_1/sdaccel_doc/hls-pragmas-okr1504034364623.html#sxx1504034358866.

[48] Xilinx. 2019. Vivado hls 2019.1: pragma hls data_pack. https://www.xilinx.com/htmldocs/xilinx2019_1/sdaccel_doc/hls-pragmas-okr1504034364623.html#drx1504034362276.

[49] Xilinx. 2019. Vivado hls 2019.1: pragma hls unroll. https://www.xilinx.com/htmldocs/xilinx2019_1/sdaccel_doc/hls-pragmas-okr1504034364623.html#uyd1504034366571.

[50] Xilinx. 2023. Vitis high-level synthesis user guide (ug1399): quantization modes. https://docs.xilinx.com/r/en-US/ug1399-vitis-hls/Quantization-Modes.

[51] Martín Abadi et al. 2015. TensorFlow: large-scale machine learning on heterogeneous systems. Software available from tensorflow.org. (2015). https://www.tensorflow.org/.

[52] Xilinx. 2023. Overview of arbitrary precision fixed-point data types. https://docs.xilinx.com/r/en-US/ug1399-vitis-hls/Overview-of-Arbitrary-Precision-Fixed-Point-Data-Types.

[53] Raghuraman Krishnamoorthi. 2018. Quantizing deep convolutional networks for efficient inference: A whitepaper. *CoRR*, abs/1806.08342. http://arxiv.org/abs/1806.08342 arXiv: 1806.08342.

[54] Benoit Jacob, Skirmantas Kligys, Bo Chen, Menglong Zhu, Matthew Tang, Andrew G. Howard, Hartwig Adam, and Dmitry Kalenichenko. 2017. Quantization and training of neural networks for efficient integer-arithmetic-only inference. *CoRR*, abs/1712.05877. http://arxiv.org/abs/1712.05877 arXiv: 1712.05877.

[55] Xilinx. 2021. Xilinx runtime (xrt) library: query command. https://docs.xilinx.com/r/2021.2-English/ug1393-vitis-application-acceleration/query.

[56] Xilinx. 2023. Alveo u50 data center accelerator card/buy online from amd. https://www.xilinx.com/products/boards-and-kits/alveo/u50.html#buy-from-xilinx.

[57] SHI. 2023. Nvidia tesla v100 - gpu computing processor overview. https://www.shi.com/product/34625444/NVIDIA-Tesla-V100-GPU-computing-processor.

[58] Timothy Dozat. 2016. Incorporating nesterov momentum into adam. In *International Conference on Learning Representations*.

[59] Leslie N. Smith. 2017. Cyclical learning rates for training neural networks. (2017). arXiv: 1506.01186 [cs.CV].

[60] Xilinx. 2023. Alveo u250 data center accelerator card. https://www.xilinx.com/products/boards-and-kits/alveo/u250.html.

[61] Xilinx. 2023. Xilinx dsp48e2 block. https://docs.xilinx.com/r/en-US/ug958-vivado-sysgen-ref/DSP48E2.