TA5 WP4-2 report
Generic Tools for Artificial Neural Network Implementation on Field Programmable Gate Arrays

A.B. Cee[1], Arno Straessner[10], and Johann C. Voigt[10]

[1]Universität Efg
[10]Technische Universität Dresden

Abstract

Text

Contents

1 Introduction

General content of the sections:

- Focus on hardware and firmware/HLS solutions, findings, experience, results
- reference to code repository (ideally public)
- only references to ANN training and training results

Here some text on importance of fast feature extraction in data flow of physics experiments, FPGA solutions and ANN approaches. Connection to PUNCH4NFDI [1] as future service provider.

1.1 Relation to other work in TA5

2 Evaluation of hls4ml for real-time classification of astronomical radio signals

editors: MPIfR group

2.1 Background

With large radio arrays such as MeerKAT and SKA becoming the primary radio facilities, these telescopes generate enormous data rates of up to tens of terabits per second. Due to the difficulty of storing such massive data volumes, real-time processing is essential. However, the radio data are often contaminated by radio frequency interference (RFI) caused by human activities. This contamination results in numerous false-positive candidates, particularly in pulsar and fast radio burst (FRB) searches. Traditional methods often struggle to effectively identify these signals, making machine learning (ML) techniques increasingly necessary.

Furthermore, in transient signal searches, only a small fraction of the data contains actual signals. Real-time search capabilities enable trigger modes, which allow the preservation of useful data and facilitate follow-up observations. To support this functionality, low-latency, high-throughput ML-based classifiers are required.

In this project, we conducted experiments to implement an ML-based classifier on an FPGA using the HLS4ML framework. The details of this implementation are presented below.

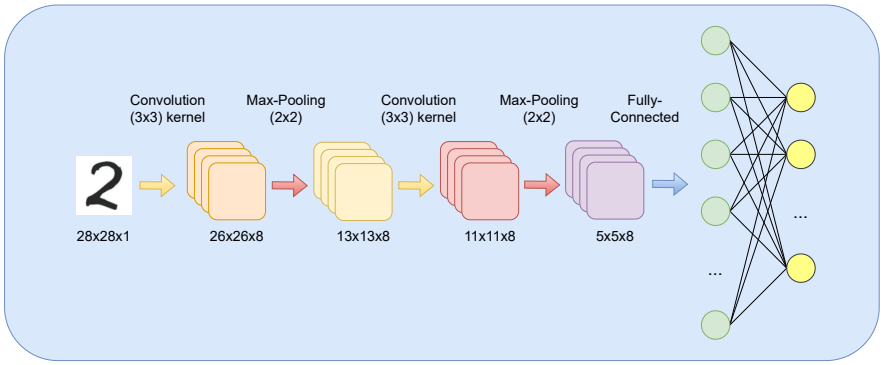

Figure 1: Diagram of the CNN-based classifier for evaluating the HLS4ML framework.

2.2 FPGA Solutions

We evaluated the HLS4ML framework by deploying a CNN-based classifier trained on the MNIST dataset [2] onto an Alveo card (`https://github.com/ypmen/punch_workshop`). The results were compared among inferences from the original model, HLS code, and hardware output, all of which demonstrated similar accuracy. The inference latency per image was approximately $\sim$40 μs, with a throughput of up to 20,000 images per second, utilizing about 5% of the FPGA's resources. Details of the evaluation are provided below.

2.2.1 ML model

To test the HLS4ML framework, we implemented a QKeras-based classifier for the MNIST dataset. The model takes inputs of handwritten digits and outputs the probabilities for each digit from 0 to 9. QKeras enables the implementation of CNN models with quantized parameters, allowing for quantization-aware training—a modern technique that reduces computational complexity without significantly impacting model accuracy. The model contains approximately 2,700 parameters in total.

This approach is particularly useful for implementing ML models on FPGAs, as FPGAs natively support fixed-point formats, which can significantly reduce resource usage compared to floating-point formats. To further minimize computational demands, we applied pruning to the model, reducing the number of parameters by a factor of 2.

The pruned QKeras model was trained on a CPU, achieving an accuracy of over 95%. This trained model was then used as input for the HLS4ML framework.

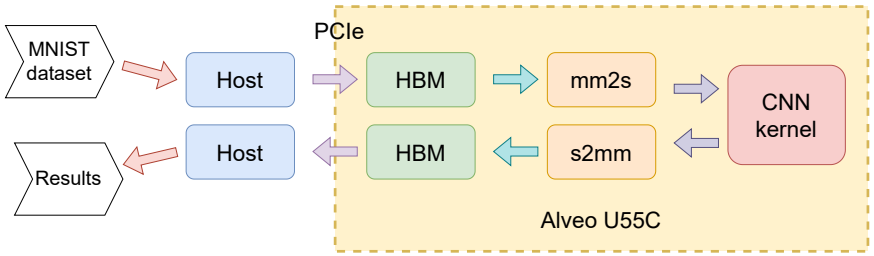

Figure 2: Diagram of the testbench of the ML kernel on FPGA.

2.2.2 HLS4ML framework

HLS4ML is a framework that converts TensorFlow models into Vitis HLS C++ code. This HLS code can subsequently be converted into HDL and synthesized using Xilinx toolchains. The HLS4ML framework requires configuration for each CNN layer. To ensure that the output HLS code aligns with the original QKeras model, we set the precision in the HLS4ML configuration to match that of the input model. The framework has two limitations:: (1) The size of the filter, i.e. $n_{\text{height}} \times n_{\text{width}} \times n_{\text{channel}} \times n_{\text{filter}}$, must be less than 4096 to enable loop unrolling during compilation; (2) The input size of the dense layer cannot exceed 4096.

Using the HLS4ML framework, the QKeras model was transformed into HLS code. The resulting HLS code was verified as a standard C++ library by testing it on the same input dataset. The accuracy of the HLS code output matched that of the original QKeras model.

2.2.3 FPGA Implementation

To implement the model on an FPGA, we exported the HLS code generated by the HLS4ML framework as a Xilinx Vitis kernel. This kernel was then built using the Vitis toolchain to generate a bitstream. For demonstration, we used the Xilinx Alveo U55C card in our experiment. The testbench operated as follows:

1. The test dataset was transferred from the host to the High Bandwidth Memory (HBM) on the Alveo card via the PCIe bus;
2. The data were fed into the ML kernel for inference;
3. The inference results were written back to the HBM;
4. The results were transferred from the Alveo card back to the host for verification.

As a result, the inference outputs from the FPGA matched those obtained from the CPU.

Table 1: Summary of Utilization Estimates on Alveo U55C card.

Name	BRAM_18K	DSP	FF	LUT	URAM
Total	**80**	**14**	**18605**	**76663**	**0**
Available SLR	1344	3008	869120	434560	320
Utilization SLR (%)	5	0	2	17	0
Available	4032	9024	2607360	1303680	960
Utilization (%)	1	0	0	5	0

3 VHDL implementation of convolutional neural networks for real-time processing of ATLAS Liquid-Argon Calorimeter data

editors: Dresden group

3.1 Overview of LAr calorimeter off-detector upgrade

Starting in 2026, the ATLAS detector at CERN will undergo an upgrade in preparation for the high-luminosity phase of the LHC. As part of this upgrade, 556 Intel Agilex-7 FPGAs will be installed for real-time processing of the liquid argon (LAr) calorimeter signals. They will be installed in a cavern shielded from the radiation environment at the detector. One FPGA will receive the data of up to 384 detector cells via 66 optical links and send the processed data towards the ATLAS trigger system and the readout.

3.2 Requirements and their influence on network training and firmware design

The LAr calorimeter produces pulses that are significantly longer than the time until new events can trigger another signal. This means that it is possible for the pulses from multiple events to overlap. This problem increases under the running conditions forseen for the high luminosity phase of the LHC. To better deal with this behavior, small artificial neural networks (ANNs) are being evaluated as a potential digital signal filter. The main purpose of this filter is to determine the energy deposited per detector cell.

The input data for one such network will be the 40 MHz ADC output of one detector cell. One architecture suited for a 1-dimensional regression problem is a convolutional neural network (CNN). They also have a relatively simple structure, which will help with the implementation on FPGAs.

The high bandwidth of the data to process, combined with the large number of optical links, severely limited the choice of FPGA and were important factors for choosing the Intel Agilex-7 device family.

As the results are required by the ATLAS trigger, a tight latency budget of approximately 150 ns applies to the planned inference in the FPGA firmware.

This has an influence on the architecture of the network, as the number of layers should remain as low as possible to stay within this latency budget.

Since each of the 384 detector cells processed on one FPGA should be handled independently, one CNN per cell is required. To leave some margins and space for other firmware components on the FPGA, the networks should therefore not contain more than 400 multiply-accumulate operations.

The FPGA contains specialized multipliers that can be used for 2 simultaneous multiplications, if the bit width is limited to 18 bit. Therefore, quantization-aware training should be used to optimize the network for 18 bit fixed point numbers.

The goal is also to have only one compiled firmware for all FPGAs in the system. This means that the weights need to be configurable at run-time. A configuration via Ethernet using the IPBus has already proven itself in earlier LAr projects and will therefore be used again. Furthermore, it is not possible to apply optimizations to the CNN architecture, which depend on the trained weights. The common technique of pruning is therefore not applicable as it is not guaranteed that the same nodes need to be pruned for all detector cells.

3.3 Firmware implementation

In order to have the best control over the resources of the FPGA, it was decided to go for a manual implementation in the low level VHDL language for the CNN inference code. CNNs have a structure that is easy to divide into multiply-accumulate-chains. This operation is then manually assigned to chains of DSP blocks on the FPGA. The basic building block for the CNN in the firmware is a filter. The number of multiplications required per clock cycle for one filter is determined by the product of the kernel size and the number of parallel inputs. The mapping of multiplications to DSPs therefore groups the input belonging to the same kernel element, but different input streams together to one DSP if possible. For an odd number of input streams, the last one is grouped in kernel direction to always assign two multiplications to one DSP block. These are connected into chains to also make best use of the chain-adder feature of the DSP blocks. This ensures a near optimal utilization of the DSPs. The results of these chains are then added in normal FPGA logic and output towards the next layer.

The input data arrives at a frequency of 40 MHz, while the FPGA is able to run at higher frequencies. Time-domain multiplexing is a concept that makes use of this, by combining multiple input channels into a single serialized stream, which is then processed at a higher frequency. For the CNN firmware, a multiplexing factor of 12 is targeted, meaning that the firmware needs to run at 480 MHz. Because one firmware block is then processing the input from 12 detector cells cyclically, the weights need to be continuously cycled between 12 sets in the same manner. When combining this with the DSP chains, the weights are required in a non-trivial order because of the time delays from the different components. It proved to be very important to store the weights in the weights memory block in the order that is required in the firmware to make the routing

as simple as possible. When configuring the network weights this induces some additional overhead because the weights need to be reordered compared to how they are usually stored on a PC.

The network training itself is done using Tensorflow through the Keras API. A custom Python script is then responsible for reading the training output files with the architecture and weights and converting it into an architecture description file for the firmware and a weights file that can be used to load the weights to the FPGA via ethernet later. The architecture description file itself describes the CNN using VHDL constants. It is read at compile time and allows the VHDL code to be flexible in terms of the number of layers, the kernel size and dilation per layer. Further options include the possibility to concatenate the input of a higher up layer with the input of the network. The precision and position of the decimal point for the fixed-point numbers to be used per layer can be configured. All these parameters are automatically extracted from the Keras or QKeras training files to allow a smooth transition from network training on a PC into the FPGA firmware. A Python model of the quantized network inference as it is implemented on the FPGA is available to verify the results.

The low level implementation is more custom-tailored for the specific application and not directly transferable to other projects due to the limited features that are supported. It is specialized to the particular FPGA architecture and therefore not directly portable to other FPGA device families or vendors. This loss in flexibility is countered by the very efficient resource utilization for the particular use case it has been designed for.

Text[3]

4 Evaluation of AI hardware engines with AMD Versal AI

editors: Mainz group

Xilinx (now part of AMD) introduced in 2018 a new category of hardware platform, the Adaptive Compute Acceleration Platform (ACAP). It is advertised as combining the strengths of FPGAs, CPUs and GPUs into one device[4]. Compared to previous system-on-chip (SoC) generations, the programmable logic (PL, the FPGA fabric) and the processing system (PS, the ARM processors) are better integrated through a network-on-chip (NoC) and shared RAM, making it easier to distribute the workload specifically to the sub-system that is best suited to the task at hand.

Versal, the first series of ACAPs, is offered in different families: AI Core/Edge, Prime and Premium. AI Core/Edge devices and certain Premium family devices feature 'AI engines' (AIE) designed for signal processing (e.g. FIR filters) and Machine Learning (ML) tasks. These vector processors provide significant parallel computing capabilities and are also utilized as Neural Processing Units (NPUs) in AMD Ryzen 7040 and 8040 Series processors under the name

XDNA[5].
Our objective is to explore the feasibility of these AI engines for stream processing in low-latency, real-time environments, for example trigger applications in High-Energy Physics.

The AI engines are Very Long Instruction Word (VLIW) processors arranged in a two-dimensional array external to the programmable logic fabric. Optimised for vector SIMD (Single Instruction, Multiple Data) processing, they are able to perform up to 32 multiply-accumulate operations in 16-bit precision in a single cycle. Unlike the DSP blocks (Digital Signal Processing) in the FPGA fabric, which are simple multiply-accumulators, the AI engines are full-fledged processors and capable to handle conditional statements, loops, arithmetic operations, and memory access. Furthermore, the engines are designed with a stream-oriented architecture, utilizing shared memories and a cascade bus to facilitate efficient data transfer between engines.

The AI engine array architecture facilitates data access and communication through several interfaces. Each engine is equipped with its own local memory. These memories are shared with the horizontal and vertical neighbours. A stream bus, constructed using AXI switches, supports static routing with single-cycle latency per switch. This bus incorporates advanced features such as broadcasting, de-/multiplexing, and Direct Memory Access (DMA), which streamline data flows within the array. Additionally, a cascade bus from an engine to one neighbour enables the efficient forwarding of intermediate results. The AI engine array has also interfaces with the Network-on-Chip (NoC) and Programmable Logic (PL) for data in- and output.

The recommended approach for utilizing AI engines is through the Vitis AI software framework, which integrates with the Deep Learning Processor Unit (DPU), an IP core. Vitis AI translates neural network models into tasks executed by the DPU, leveraging AI engines or DSPs in the PL. However, tests performed as part of the evaluation, revealed significant limitations for low-latency applications. For instance, even a trivial neural network comprising a single neuron incurs a latency of 30 µs, making it unsuitable for time-critical trigger scenarios.

An alternative to relying on the Vitis AI framework is the direct programming of AI engines. This involves the following workflow:

- PL Firmware Development: Create the programmable logic design, including the CIPS (Controller for Integrated Platform Services) and AI engine blocks (black boxes).
- Interface Definition: Define AXI-stream interfaces between the PL and AI engines, with connectivity established later in Vitis.
- AI Engine Software: Write kernels in C++ and define an AI engine graph to connect PL streams to AI engine kernels and between kernels.
- Design Compilation: Compile AI engine kernels, synthesize and implement the PL design in Vivado, and integrate all components into bootable system files.

The PL interface for AI engines supports 37 interfaces, each offering 8x 64-bit PL-to-AIE buses and 6x 64-bit AIE-to-PL buses, with a maximum interface frequency of half the AI engine clock rate. Measured latency for a round-trip operation, including a simple increment, is 51.2 ns at 625 MHz.

An example application of AI engines is a matrix-vector multiplication. A single AI engine computes the product of a 64x16 matrix with a 16x1 vector (16-bit), requiring 1,024 multiplications, in 32 cycles corresponding to a latency of 25.6 ns at a 1.25 GHz clock rate.

Various architectural strategies can be employed to optimize the mapping of applications to AI engines:

- Single Engine: Processes one event at a time, leading to high latency for multiple events.
- Round-Robin Scheduling: Distributes the workload across engines for parallel processing, reducing latency per event.
- Pipeline Architecture: Processes data sequentially across engines, with intermediate results forwarded downstream.
- Zipper Merging: Enhances pipeline designs by feeding additional data to downstream engines, increasing bandwidth but requiring careful load balancing.
- Column/Row Distribution: Suitable for applications like convolutional neural networks (CNNs), distributing computation across neighboring engines.

In conclusion, AMD Versal AI engines provide substantial parallel computing power. However, achieving their maximum performance and effectively utilizing them in low-latency stream processing designs necessitates direct connections to the programmable logic and manual programming, rather than relying on AMD's workflow involving their DPU. This requires a careful design and knowledge of the processing time of each step to avoid stalls or even deadlocks.
Our evaluation is ongoing; however, our preliminary findings suggest that the usage of AI engines for example in trigger applications is feasible.

5 Recommendations for users and developers

editors: all

- Useful public repositories and libraries
- Advice for vendor-specific IP cores and interfaces

6 Summary and Outlook

Acknowledgements

This work was funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) – project number 460248186 (PUNCH4NFDI). Special thanks to all involved PUNCH4NFDI members.

References

[1] The PUNCH4NFDI Consortium. PUNCH4NFDI consortium proposal, September 2020. This is the version documenting the work plan at the proposal stage. The reduction in funding led to a re-shaping of the work programme that is documented elsewhere. doi:10.5281/zenodo.5722895.

[2] Yann LEcunn and Corinna Cortes. The MNIST Database, 2023. available at https://yann.lecun.com/exdb/mnist/.

[3] Georges Aad, Anne-Sophie Berthold, Thomas Calvet, Nemer Chiedde, Etienne Fortin, Nick Fritzsche, Rainer Hentges, Lauri Laatu, Emmanuel Monnier, Arno Straessner, and Johann Voigt. Artificial neural networks on fpgas for real-time energy reconstruction of the atlas lar calorimeters. *Computing and Software for Big Science*, 5, 12 2021. doi:10.1007/s41781-021-00066-y.

[4] Xilinx, Inc. Versal, the first adaptive compute acceleration platform. URL: https://docs.amd.com/v/u/en-US/wp505-versal-acap.

[5] Advanced Micro Devices, Inc. AMD XDNA™ Architecture. URL: https://www.amd.com/en/technologies/xdna.html.