Neural Networks on FPGAs for Real-Time Data Processing

Opportunities, Implementation, and a Case Study

Tobias Habermann[1] Martin Kumm[1] Rahul Singh[2]

IBIC 2026

[1]Hochschule Fulda - University of Applied Sciences

[2]GSI Helmholtzzentrum für Schwerionenforschung

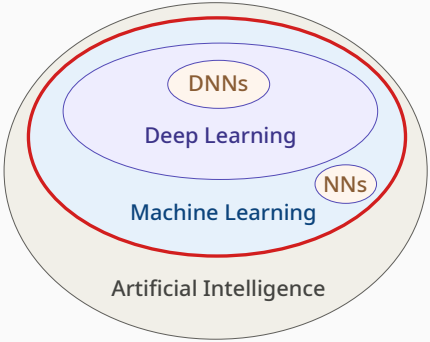
Outline

1. Introduction
2. Motivation
3. Overview of the approach
4. Case Study: Real-Time Particle Counting at GSI
5. Implementation on the FPGA
6. Identifying Use Cases: When can AI be the right tool?
7. Outlook

Introduction

Artificial Intelligence, Machine Learning, Deep Learning

Artificial Intelligence (AI) is a broad spectrum that spans over all the algorithms used to emulate intelligent behavior.

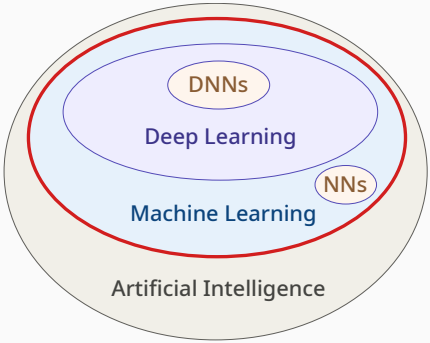


Machine Learning (ML)

- Relies on statistical methods that learn from data and generalize to unseen inputs.
- Learning is performed by optimizing a loss function, rather than following hand-coded rules.

Artificial Intelligence, Machine Learning, Deep Learning

Machine Learning (ML) is commonly implemented using Neural Networks (NN), models loosely inspired by the biological brain.



In this talk we focus on Neural Networks (NN)

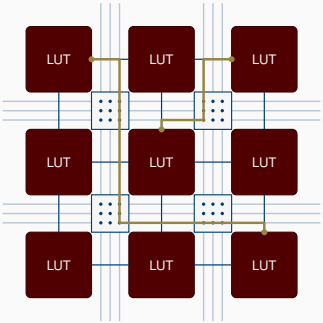
- Neural Networks consist of layers of connected units (neurons) that jointly learn a function.
- Stacking many layers to get a Deep Neural Network (DNN)
- You get a Convolutional Neural Network (CNN) when you use convolutional layers instead (more later).

What is an FPGA?

Field Programmable Gate Array (FPGA) is a device that can implement arbitrary digital circuits.

How does that work?

1. The FPGA contains:
   - Programmable Lookup Tables (LUTs)
   - ROM/RAM blocks (BRAMs)
   - Digital Signal Processor blocks (DSPs)
   - Flip-Flops (FFs)
2. The interconnections can be programmed too!



This allows us to *program* a circuit onto the FPGA fabric!

Motivation

AI in Beam Instrumentation

AI is already used in beam instrumentation.

Examples

- Setup & calibration automation
- Data-quality monitoring
- Offline analysis of recorded data

Often no real-time processing constraints.

Easy to solve with CPUs/GPUs!

AI in Beam Instrumentation

But AI *may* help, in other scenarios too!

What about scenarios where ...

- Sensor data is hard to interpret due to hardware limitations
- Events are too complex for traditional methods to catch
- Detecting a precursor reliably *and* in time is critical
- Filtering is mandatory due to sheer data volume, but too complex

AI may help, or is already helping here, but it is not running fast enough!
In that case → **FPGAs are the perfect platform!**

Example: Anomaly Detection at the CMS L1 Trigger

The LHC collides protons at 40 MHz; the CMS detector has ˜100 million readout channels.[1]

The constraint

- Custom FPGAs (the L1 Trigger) reduce 40 MHz down to ˜100 kHz within a fixed 4 µs latency.
- Selection relies on predefined, rule-based criteria (e.g. "two jets above X GeV").

The problem

- Simple filters may not catch unexpected new physics signatures.
- Filtered events are discarded forever – no second chance offline.

The decision must be right the first time, and made within 50 nanoseconds.

[1] A. Gandrakota (CMS Collaboration), "Real-time Anomaly Detection at the L1 Trigger of CMS Experiment," PoS ICHEP2024, 1025 (2024)

Example: Anomaly Detection at the CMS L1 Trigger

The AI approach

- An autoencoder is trained only on ordinary collisions – it learns to reconstruct "normal" events well.
- Anomalous / new-physics events reconstruct poorly $\Rightarrow$ high reconstruction error = anomaly score.
- Model-independent and unsupervised: no need to predefine what new physics looks like.

Implementation

- The ML model inference was implemented on an FPGA.
- Inference fits within a 50 ns latency budget, directly in the trigger data path.

Tested live in 2024: stable operation, +46% efficiency for rare signatures at equal trigger rate.

Example: Real-Time MHD Mode Tracking on a Tokamak

Rotating plasma instabilities on the HBT-EP tokamak can grow and disrupt the discharge.[2]

The constraint

- Modes rotate at ˜10 kHz – control must react several times per rotation.
- Classic diagnostic: dozens of in-vessel magnetic sensors wired into the machine.

An alternative: look, don't touch

- Swap wired sensors with cameras
- They use high-speed optical cameras (100+ kfps)
- thousands of pixels per frame have to be processed

100k images per second. Processing each image within a few microseconds

[2] Y. Wei et al., "Low latency optical-based mode tracking with machine learning deployed on FPGAs on a tokamak," Rev. Sci. Instrum. 95, 073509 (2024)

Example: Real-Time MHD Mode Tracking on a Tokamak

The AI approach: a CNN reading camera frames

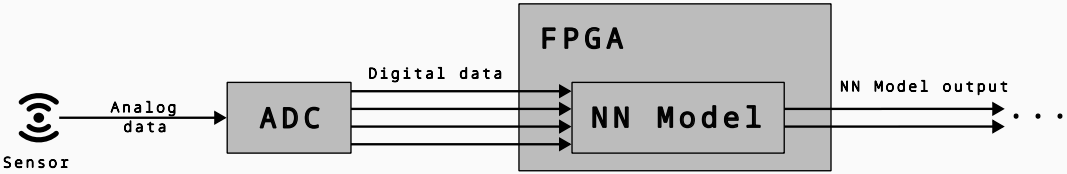
- A small CNN predicts the mode's sine/cosine components directly from each camera image
- more accurate than the non-ML baselines tested (linear regression, SVD).

Implementation

- Deployed directly on the FPGA already onboard the commercial camera frame-grabber card.
- Total trigger-to-output latency 17.6 µs (7.7 µs for the CNN itself), at up to 120 kfps.

First real-time deep-learning MHD mode tracker on a tokamak – comparable latency to the earlier magnetic-sensor system, from camera images alone.

FPGAs for Real-Time Inference



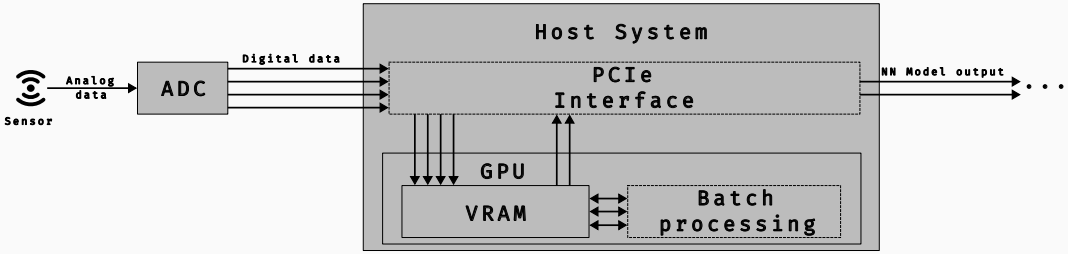
FPGAs are the perfect platform for real-time inference/prediction!

FPGAs

- A fixed pipeline allows deterministic latency
- Can be placed directly in the data path of a digitizer. So no host round-trip, and it sits directly at the sensor.
- *But:* only if the circuit is designed to keep up with the incoming data rate, every clock cycle.

That design challenge is what this talk is about.

FPGAs for Real-Time Inference

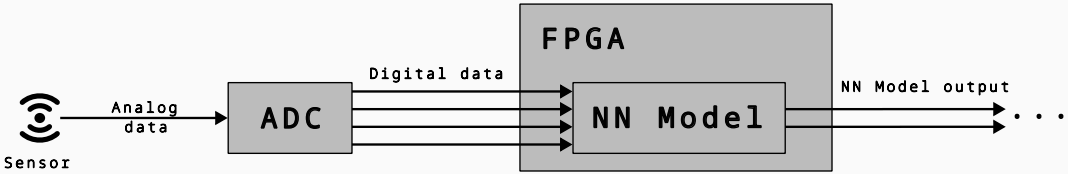


CPUs / GPUs

- Throughput comes from batching – which adds latency and jitter
- OS scheduling, PCIe transfers, memory hierarchy: non-deterministic
- Not designed to sit *inline* in a fast data path

Overview of the approach

A simple setup for inline processing



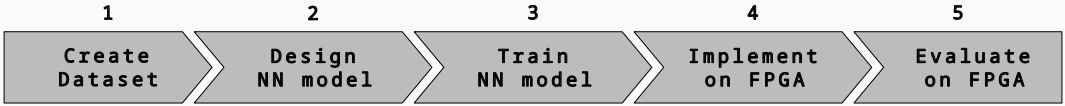
The end result is a solution that is processing data inline.

- Connect the FPGA *directly* to the ADC output.
- Process data inline without batching.
- Implement the inference of the NN model as a circuit on the FPGA.

This allows for a low latency implementation without jitter!

From Dataset to FPGA Implementation

This is how we get there!



Case Study: Real-Time Particle Counting at GSI

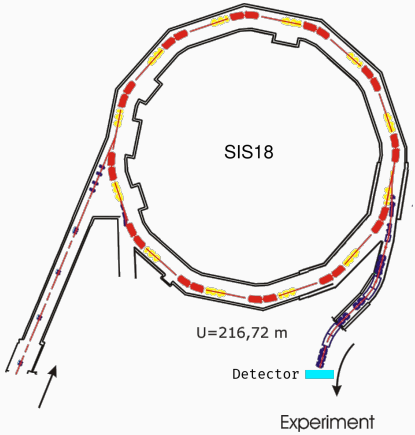
The GSI Beam Instrumentation Use Case: Pile-Up in Single-Particle Counters

The Goal

- Count single particles.
- Determine arrival times of single particles (Localization).

The Plan

1. Place a detector in the beam.
2. Particles hitting the detector create a pulse signal.
3. Localize each individual pulse in the signal.

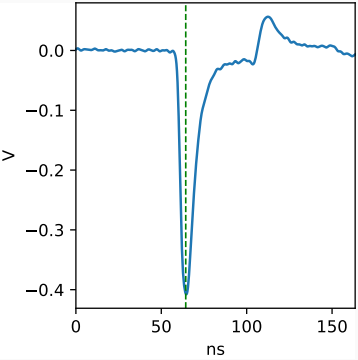


A ring accelerator with a scintillator (light blue) placed at the output.

The GSI Use Case: Pile-Up in Single-Particle Counters

The Setup

- Scintillator (BC400) + PMT at the extraction beamline
- Carbon ions (C^{6+}) at 300 MeV/u from the SIS18 synchrotron
- Signal digitized at 10 GSa/s, 12-bit, on a COTS digitizer with an on-board FPGA

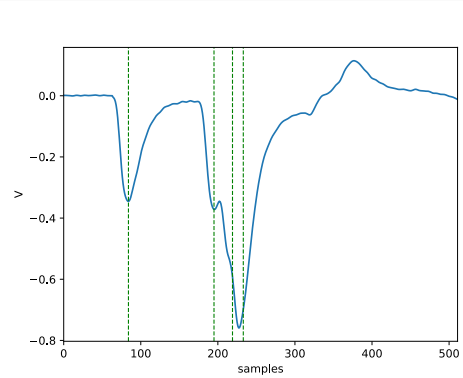


The pulse generated by a single particle.

The GSI Use Case: Pile-Up in Single-Particle Counters

The Problem: Pile-Up

- At high extraction rates, multiple particles arrive almost simultaneously
- Pulses overlap into a composite waveform
- Under- / Overcounting (spill fluctuations) propagates directly into downstream physics results[3]

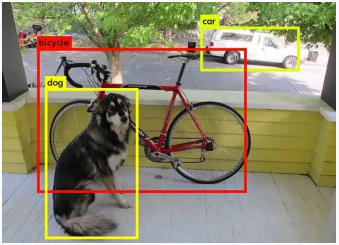


The signal when three particles arrive at almost the same time. A pile-up composed of three overlapping pulses.

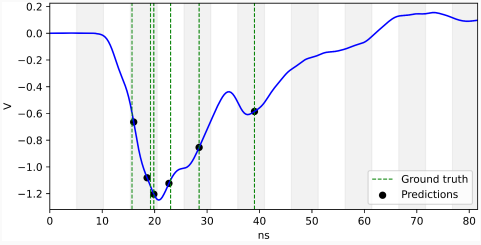
[3] P. Niedermayer et al., "Spill optimization system improving slow extraction at GSI," JACoW IPAC2025, TUPB008 (2025)

PiLoNet: A YOLO-Inspired 1D CNN

Pile-up **Lo**cator **Net**work (PiLoNet)[4] – Inspired by YOLOv1[5], adapted to 1D time series



The output prediction of a YOLO box detector model.



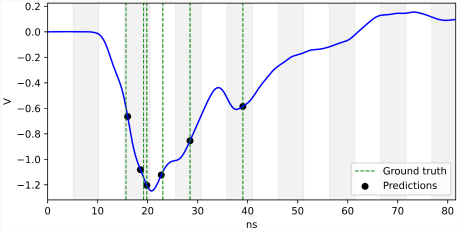
Predicted (dots) vs. real pulse positions (dashed lines).

[4] T. Habermann, M. Kumm, R. Singh, "Application of Convolutional Neural Networks for Pile-Up Correction in Single-Particle Counting," JACoW IBIC2025, MOPCO36 (2025)

[5] J. Redmon et al., "You Only Look Once: Unified, Real-Time Object Detection," CVPR (2016)

PiLoNet: A YOLO-Inspired 1D CNN

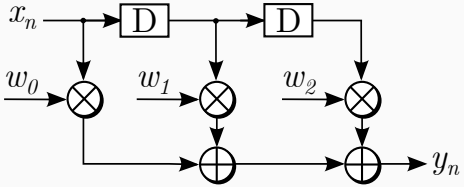
- Input window divided into equal-sized cells – each cell spans 32 consecutive input samples ($\sim$3.2 ns at 10 GSa/s)
- Each cell independently predicts up to m pulses: confidence score + relative position
- A pulse is predicted when confidence > 0.5
- 6 convolutional layers, **no** fully connected layers



Predicted (dots) vs. real pulse positions (dashed lines).

The principal of the convolutional layer

From a signal processing POV, a convolutional layer is composed of *trainable FIR filters*.

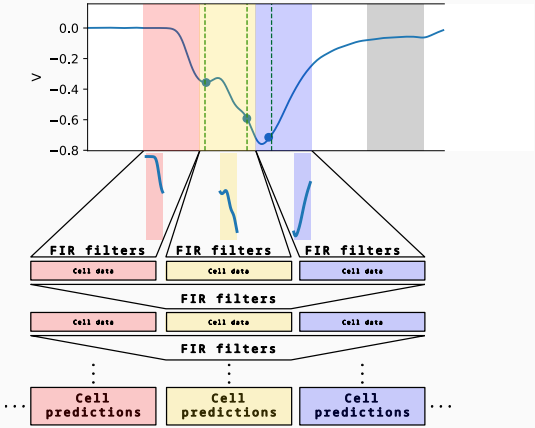


The core operation in convolutional layer. The example has a kernel size of 3

- N-tap filter where N is set to the kernel size.
- x_n is the input and y_n the output signal.
- All w_i are the coefficients.
- A non-linear operation follows after it (left out for simplification)

PiLoNet: A YOLO-Inspired 1D CNN

1. The input is divided into equal-sized cells.
2. After that, each layer is processing each cell.
3. Each cell output is based on the previous cell data, and their adjacent cell data using a 3-tap filter.



A simplified visualization that shows how the CNN works

Data Collection & Training

Data collection

- Collected at the HTP beamline, GSI
- Algorithmic peak detection + manual correction: $\sim$26,000 labeled pulses

Create synthetic pile-ups

- Synthetic pile-ups via linear superposition of real single pulses
- This fixes class imbalance and gives exact peak locations

Training process

1. Split data into training and validation data
2. Train the model on the training data
3. Validate using the validation data

Performance: CNN vs. Traditional Methods

How to measure the performance?

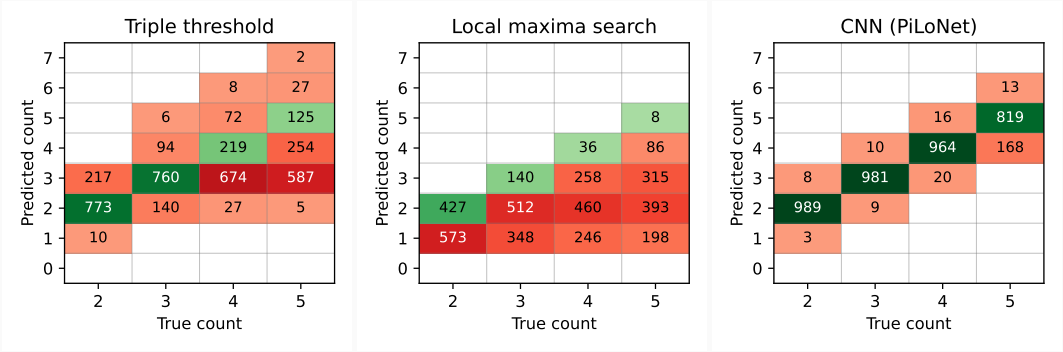
- We need to know the *precise* location of each pulse.
- We have to match each prediction to the actual particle location.

So what we did was:

- Created synthetic pile-ups again (using the single pulses from the validation dataset).
- Disregarded results, per pile-up, where the predicted number does not match the true number of particles to avoid ambiguity.

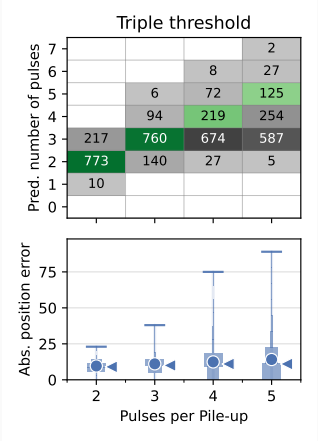
Dataset: 4000 pile-up windows, 1,000 each with 2, 3, 4, and 5 overlapping pulses. So 14,000 pulses in total.

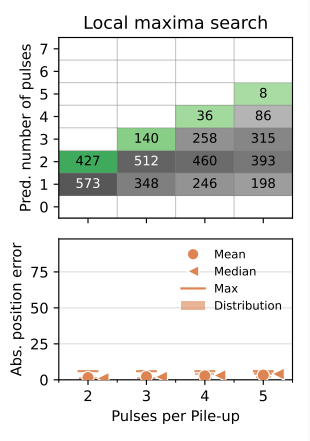
Performance: CNN vs. Traditional Methods

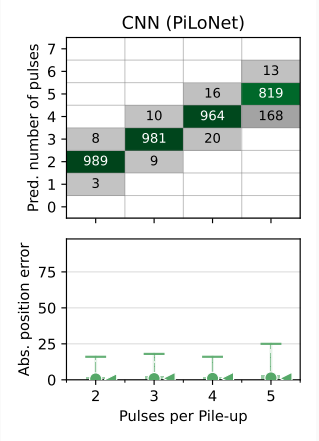


	Triple thresholding	Local maxima search	**PiLoNet (CNN)**
Failed pile-ups (of 4,000)	2,123 (53.1 %)	3,389 (84.7 %)	**247 (6.2 %)**

Performance: CNN vs. Traditional Methods

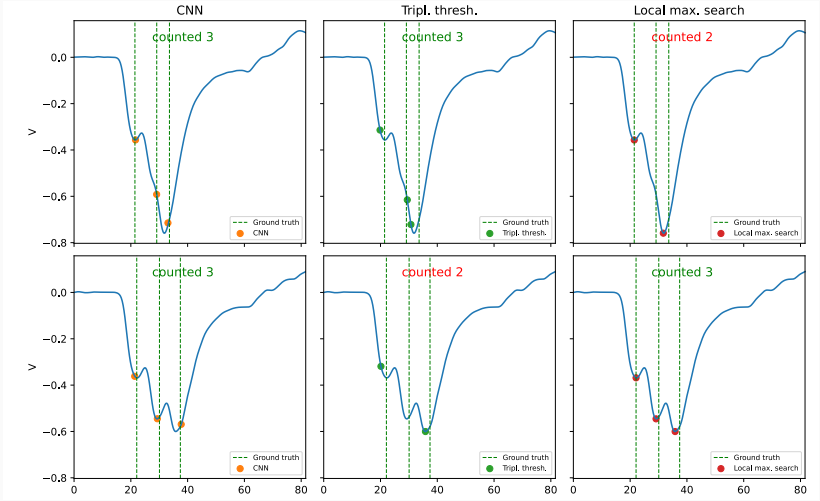






	Triple thresholding	Local maxima search	**PiLoNet (CNN)**
Average loc. err.	11.236	1.913	**0.994**
Median loc. err.	10.000	**1.000**	**1.000**

CNN vs. Traditional Methods



Two examples (one per row) where all three methods are compared.

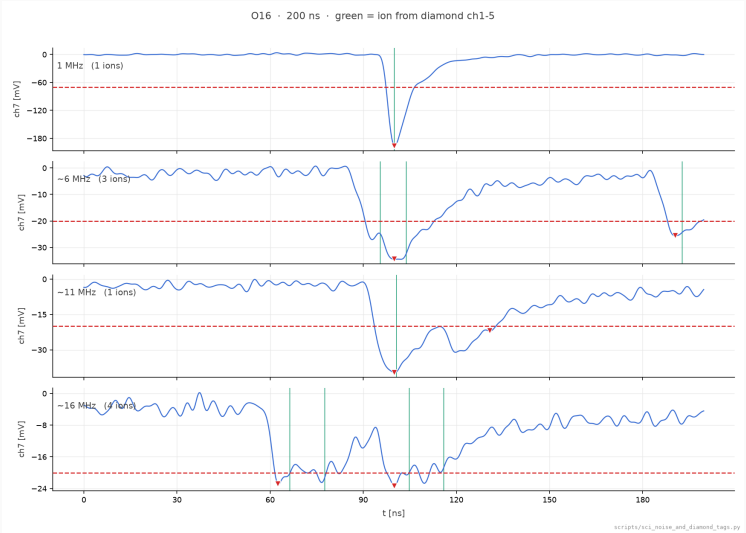
Why Traditional Methods Fail

- **Threshold crossing:** misses pulses which are more spread out
- **Local maxima search:** fails in dense pile-up

Additional problems:

- High count rates droop the PMT dynode voltage, compressing gain and distorting pulse amplitudes
- The scintillator's slow decay tail creates a floating baseline, shifting the amplitude of the following pulse
- The detector response can become non-linear when pile-up occurs

Why Traditional Methods Fail: Example



A low rate, narrow pulse with -180 mV amplitude. At higher rate the amplitude is reduced to -20 mV and the pulses have become broader (PMT sag)

FPGA Implementation

Platform

- On-board Xilinx Kintex UltraScale FPGA
- ∼5,000 DSP slices, ∼600k LUTs, ∼2,000 BRAM blocks
- Input: 10 GSa/s, 12-bit

Teledyne SP Devices ADQ35 digitizer

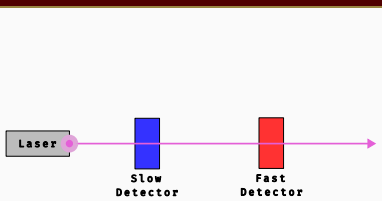
	LUT	FF	BRAM	DSP
Used	25,987	54,136	0	833
Util.	4 %	4 %	0 %	15 %

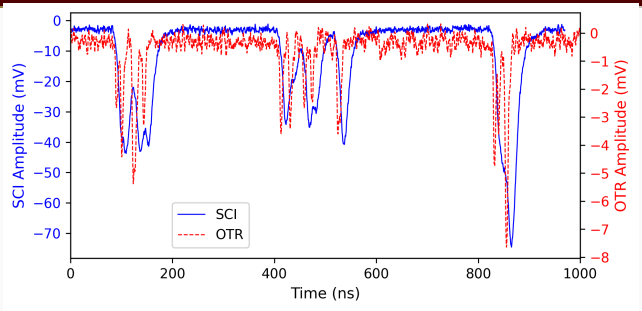
Substantial headroom remains! So a lot of room for larger or more complex models.

Further Information

- Latency 291.2 ns
- Running at 312.5 MHz
- Processing 32 samples per clock cycle

Validation Against a Real Spill





Recorded scintillator (SCI, blue) and fast OTR reference (red) waveforms.[6]

- Recorded scintillator waveform from a non-optimized spill, 20 MHz average rate
- Played back via signal generator at 10 GSa/s (512 cells, 16,384 samples)
- CNN: All pulses correctly identified!
- Single threshold: Only 8 of 15 pulses found at its best threshold

[6] R. Singh et al., "CNN-Based Arrival Time Determination in Piled-Up Particle Counter Signals," JACoW IPAC2026, WEP6047 (2026)

Implementation on the FPGA

Implementation on the FPGA

How to implement NN models on FPGAs?

- We need a circuit that implements the inference/prediction

Learnings so far:

- **There is no pause.** The circuit has to process samples every clock cycle.
- **We have a constant data rate.** The circuit has to process a fixed number of samples per clock cycle.

Let's design an Implementation Framework

- **From model to circuit.** It should take a NN model, and design the circuit for it

What a Good Implementation Framework Needs

Inline processing

Guarantee that data can be processed every clock cycle.

- Design the circuit with enough processing units
- Design the processing units to be able to process data every clock cycle or in fixed periods.

What a Good Implementation Framework Needs

Scalability

Targeting a lower data rate should lead to a smaller circuit.

- The framework should be able to scale the circuit up or down
- Design the framework to support different foldings of the arithmetic operations
- Design the processing units to support different data interleavings and parallelization schemas.

What a Good Implementation Framework Needs

High accuracy

Avoid heavy quantization/pruning

- Reducing the word width too much (heavy quantization) reduces the accuracy
- Cutting out neurons or reducing the number of convolutional filters (pruning) also reduces accuracy.
- The framework should include hardware-aware training to keep a high accuracy.

What a Good Implementation Framework Needs

Resource efficiency

Use the FPGA resources efficiently.

- High-level synthesis (HLS) reduces the control you have over the circuit.
- Rather than using HLS, VHDL/Verilog should be used to allow more optimization
- Use state-of-the-art arithmetic that is optimized for FPGAs

What a Good Implementation Framework Needs

Flexibility

It should support a wide range of neural network types.

- But still keep it simple.
- A good implementation framework has just a handful of basic components.
- They have to be designed to be used as building blocks to construct other operations with them.

A Continuous-Flow Architecture

Two core ideas

Continuous flow Every arithmetic unit produces and consumes data every cycle, for any degree of parallelization – often removing buffering between layers entirely

Data-rate awareness Parallelism and data interleaving per layer are derived *automatically* from the layer parameters and the input data rate

A Continuous-Flow Architecture

Only 3 arithmetic processing units are needed

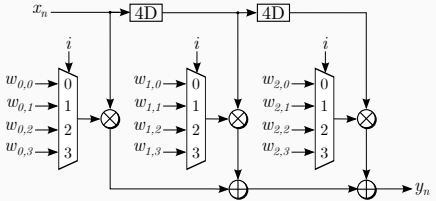
- **Kernel processing unit (KPU)** for convolutional and depthwise-convolutional layers
- **Fully connected unit (FCU)** for fully-connected and pointwise-convolutional layers
- **Pooling processing unit (PPU)** for pooling layers

Almost everything else (e.g. LSTM layers) is a combination of these; activation functions (mostly ReLU) come essentially for free.

The circuit is generated directly in VHDL, using state-of-the-art FPGA arithmetic (FloPoCo[7]).

[7] F. de Dinechin and B. Pasca, "Designing Custom Arithmetic Data Paths with FloPoCo," IEEE Design & Test (2011)

Interleaving data: KPU Example



- This KPU processes 4 different kernels, one per clock cycle – reusing the same arithmetic units
- Weights are stored in BRAM and swapped every cycle
- The degree of interleaving is derived automatically from the layer's data rate

Establish a Continuous Flow of Data in Each Layer

- How many processing units are needed?
- How to calculate the output data rate?

First: how to calculate the output data rate of a layer?

Symbol	Description
k	Kernel size
s	Stride
ℓ	Layer index
$d_{\ell-1}$, d_ℓ	Number of input/output channels of layer ℓ
$r_{\ell-1}$, r_ℓ	Input/output data rate of layer ℓ (features/cycle)
j	Number of parallel input signals
h	Number of neurons/filters processed sequentially
a	Number of pixels processed per clock cycle
C	Number of weight reconfigurations
J_ℓ, H_ℓ	Set of valid values for j, h in layer ℓ
HJ_ℓ	Set of all valid (j, h) (or (j, h, a))

$$r_0 = \frac{3}{4} \quad \Longrightarrow \quad r_\ell = \frac{d_\ell \cdot r_{\ell-1}}{d_{\ell-1} \cdot s^2} \quad \Longrightarrow \quad \text{proceed with the next layer}$$

Given: the input data rate (e.g. 3 inputs every 4 clock cycles) **Calculate:** the current layer's output data rate from the preceding layer's input rate

Implementing layers aware of the data-rate

With the data rate for each layer we can now calculate:

- The number of parallel input signals
- The number of interleaved neurons/filters
- The number of reconfigurations
- The number of pixels to process in parallel (left out for now)

Symbol	Description
k	Kernel size
s	Stride
ℓ	Layer index
$d_{\ell-1}$, d_ℓ	Number of input/output channels of layer ℓ
$r_{\ell-1}$, r_ℓ	Input/output data rate of layer ℓ (features/cycle)
j	Number of parallel input signals
h	Number of neurons/filters processed sequentially
a	Number of pixels processed per clock cycle
C	Number of weight reconfigurations
J_ℓ, H_ℓ	Set of valid values for j, h in layer ℓ
HJ_ℓ	Set of all valid (j, h) (or (j, h, a))

1. Define viable settings

$$J_\ell = \{j \in \mathbb{N} \mid j \mid d_{\ell-1}\}$$

$$H_\ell = \{h \in \mathbb{N} \mid h \mid d_\ell\}$$

2. Filter out settings that cannot keep up

$$HJ_\ell = \left\{(j,h) \middle| j \in J,\ h \in H,\ \tfrac{j}{h} \geq r_{\ell-1}\right\}$$

3. Choose the setting barely enough

$$\mathrm{BestRate} = \min\left\{\tfrac{j}{h} \middle| (j,h) \in HJ_\ell\right\}$$

Where Off-the-Shelf Frameworks Fall Short

- **No automated reconfiguration / data interleaving**
  - Reuse of arithmetic components has to be controlled manually
  - Leads to stalls, underutilization, unnecessary buffering
- **Not designed for continuous input streams**
  - Data is buffered even at the input
  - Internal components stall, waiting for the next burst
- **Rely on HLS rather than direct VHDL/Verilog**
  - More resource overhead, less control over target-specific optimization
- **Rely on pruning / heavy quantization to scale efficiently**
  - Can degrade accuracy

That is why off-the-shelf frameworks fall short here – and why we built our own.

The Toolflow

1. Train with **quantization-aware training** to match the target hardware constraints
2. **Automatically fit** the implementation (parallelism / interleaving) to the data rate of each layer
3. **Render the model directly in VHDL**, using state-of-the-art arithmetic
4. **Self-test**: a testbench checks that every layer is bit-accurate to the trained model

Identifying Use Cases: When can AI be the right tool?

Identify the Use Case

AI *may* help, in other scenarios too!

What about use cases where ...

- sensor data is difficult to interpret because of hardware limitations?
- events are complex and hard to catch with traditional methods?
- detecting the precursor of an event reliably and in time is mandatory but challenging?
- single events are expressed by a multitude of sensors and their behavior over time?
- due to sheer mass of data, data filtering becomes mandatory, but is too complex?

Identify the Use Case

Overall you can compress this down into:

AI can be the right tool when ...

- sensor data becomes messy (due to hardware limitations).
- events/precursors are too complex and difficult to detect or filter.
- traditional methods fail.

Outlook

Outlook

- **Dataset expansion:** laser-based labeling (exact ground truth for up to 4–5 pile-ups within 10 ns)
- Try other ML approaches like autoencoders
- Test the system live! (Will be done in November this year)
- Integration into the GSI spill feedback system.
- An open data repository is planned
- Test other detectors that are even faster. (Currently we play back the data at 10 GSa/s)
- Test other PiLoNet architectures (Ablation study)
- Allow automatic adjustment of the pulse width and amplitude to cater to different detectors.

References i

F. de Dinechin and B. Pasca.
Designing custom arithmetic data paths with flopoco.
In *IEEE Design & Test of Computers*, 2011.

T. Habermann, M. Kumm, and R. Singh.
Application of convolutional neural networks for pile-up correction in single-particle counting.
In *Proc. 14th International Beam Instrumentation Conference (IBIC'25)*, pages 152–155, 2025.

A. Hamdan and T. Habermann.
Training strategies for real-time particle pile-up detection networks.
In *Proc. International Particle Accelerator Conference (IPAC'26)*, 2026.

References ii

B. Li, Y. Liu, and X. Wang.
Gradient harmonized single-stage detector.
In *Proc. AAAI Conference on Artificial Intelligence*, 2019.

T.-Y. Lin, P. Goyal, R. Girshick, K. He, and P. Dollár.
Focal loss for dense object detection.
In *Proc. IEEE International Conference on Computer Vision (ICCV)*, 2017.

P. Niedermayer, H. Bräuning, R. Singh, and T. Milosic.
Spill optimization system improving slow extraction at GSI.
JACoW, IPAC2025:TUPB008, 2025.

J. Redmon, S. Divvala, R. Girshick, and A. Farhadi.
You only look once: Unified, real-time object detection.
In *Proc. IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016.

References iii


R. Singh, P. Boutachkov, T. Habermann, and M. Kumm.
Cnn-based arrival time determination in piled-up particle counter signals.
In *Proc. 17th International Particle Accelerator Conference (IPAC'26)*, 2026.


M. Yeung, E. Sala, C.-B. Schönlieb, and L. Rundo.
Unified focal loss: Generalising dice and cross entropy-based losses to handle class imbalanced medical image segmentation.
Computerized Medical Imaging and Graphics, 2022.