

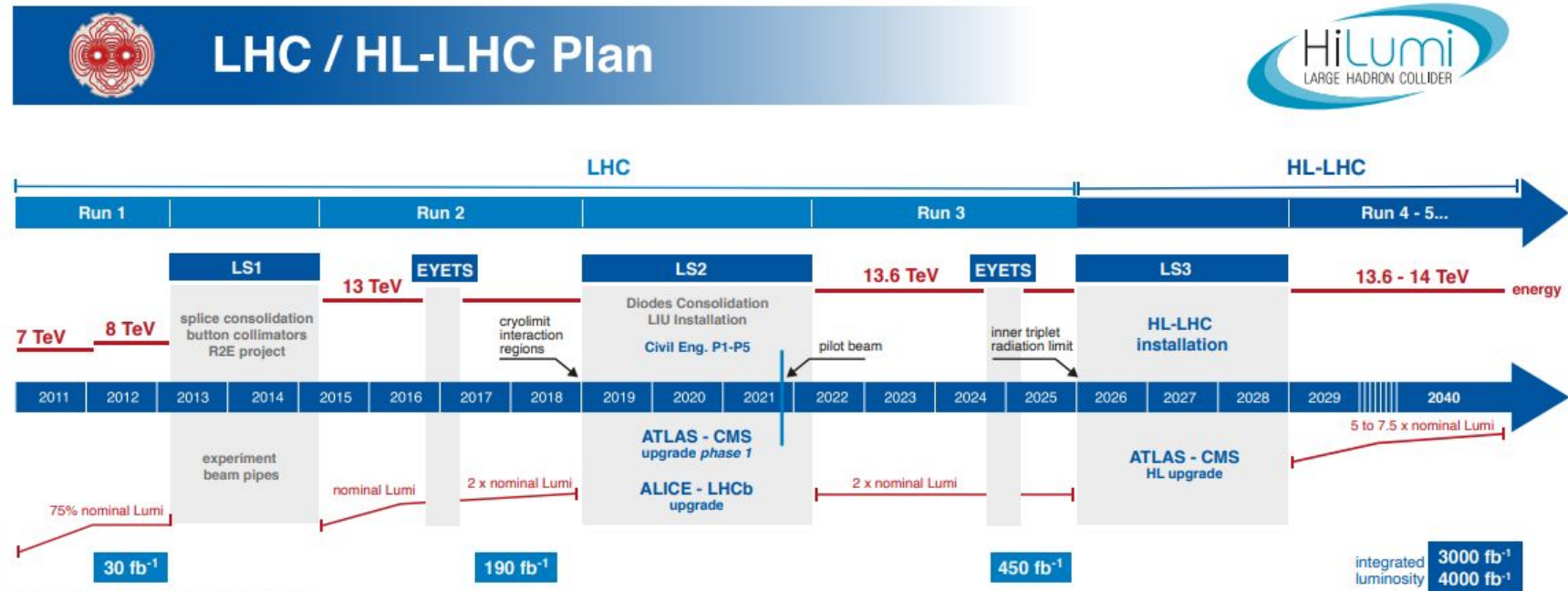
Embedded Neural Networks on FPGAs for Real-Time Computation of the Energy Deposited in the ATLAS Liquid Argon Calorimeter

TWEPP 2022 - Bergen, Norway

Georges Aad on behalf of the ATLAS Liquid Argon Calorimeter Group
CPPM, Aix-Marseille Université, CNRS/IN2P3, Marseille

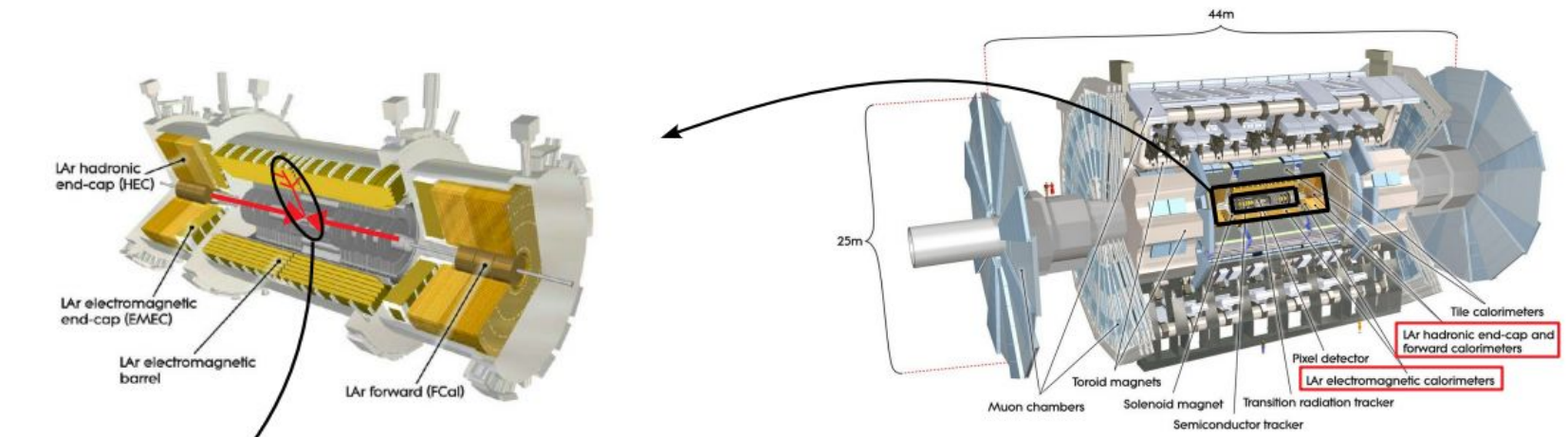
Introduction

- LHC upgrade during the long shutdown starting 2026 leading to the HL-LHC
 - Increase the instantaneous luminosity by a factor 5 to 7
 - 140 to 200 simultaneous p-p collisions (pileup)
- ATLAS will be upgraded to cope with the HL-LHC conditions
 - Increase the level 1 frequency from 100 kHz to 1MHz
 - New readout electronics for the liquid Argon calorimeter

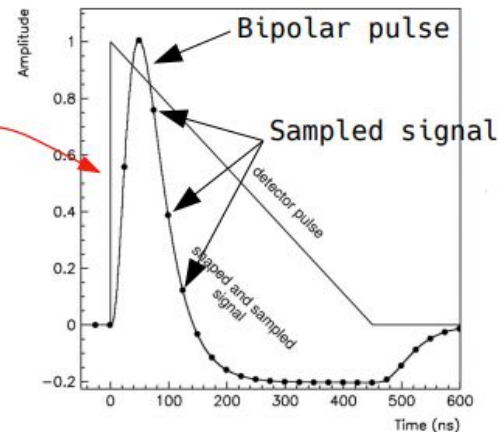
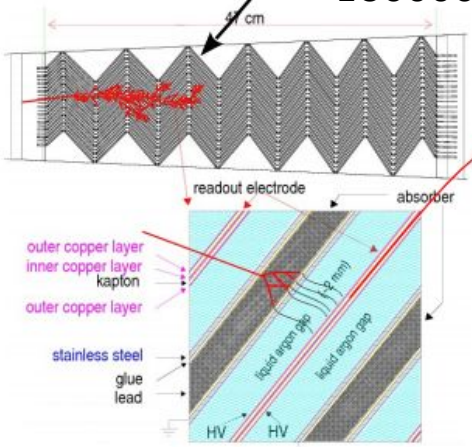


The ATLAS Liquid Argon Calorimeter

- Measures the energy of electromagnetically interacting particles mainly electrons and photons
- Trigger capabilities at the first level of triggering (implemented in hardware)
 - 40 million bunch crossings per second (2MB per event)
 - Mandatory online selection of events before permanent storage on disk



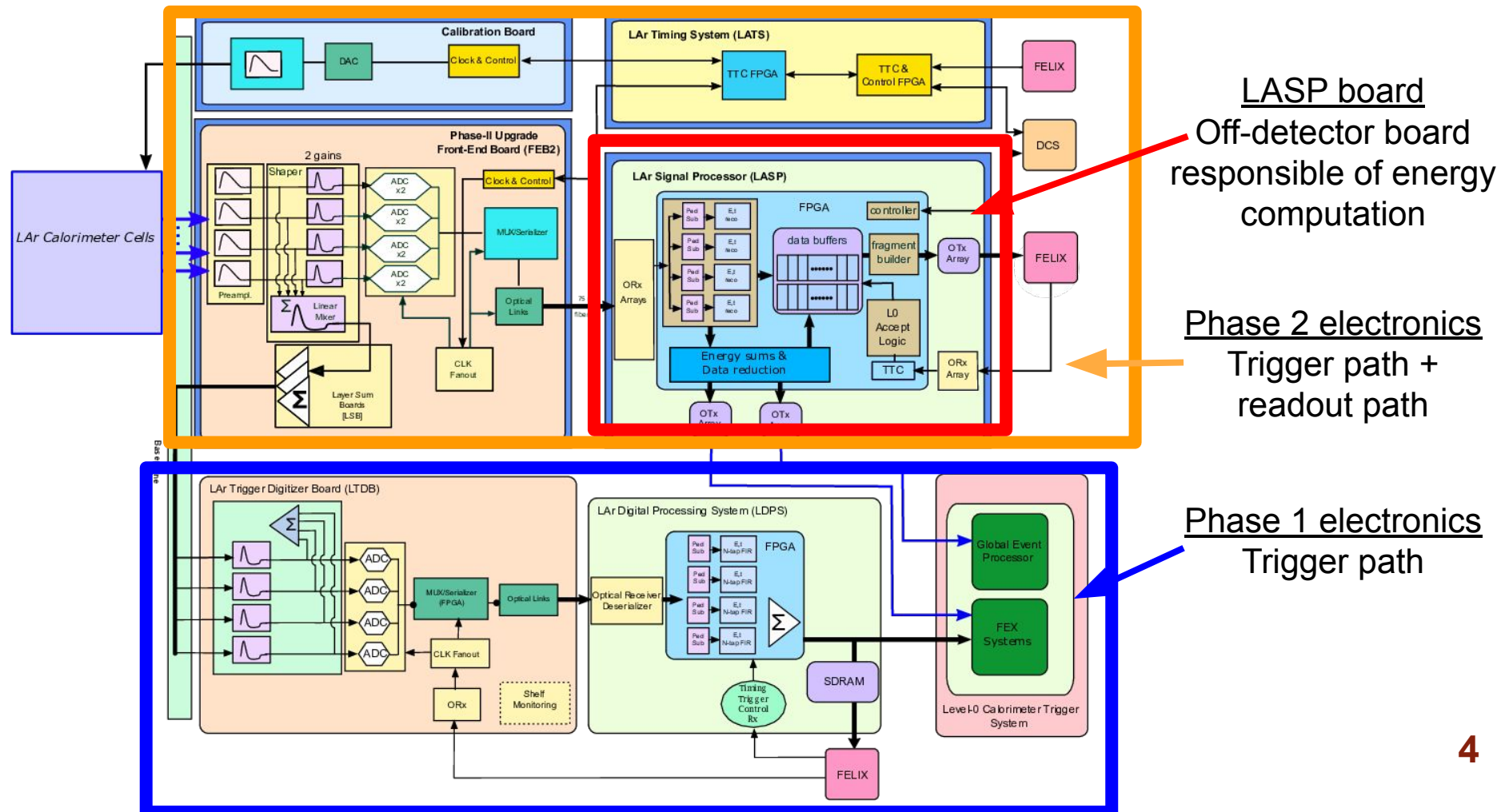
180000 channels



- Electronic signal with amplitude corresponding to the deposited energy in the calorimeter
- Shaped and sampled at 40 MHz
- Samples used to compute the deposited energy

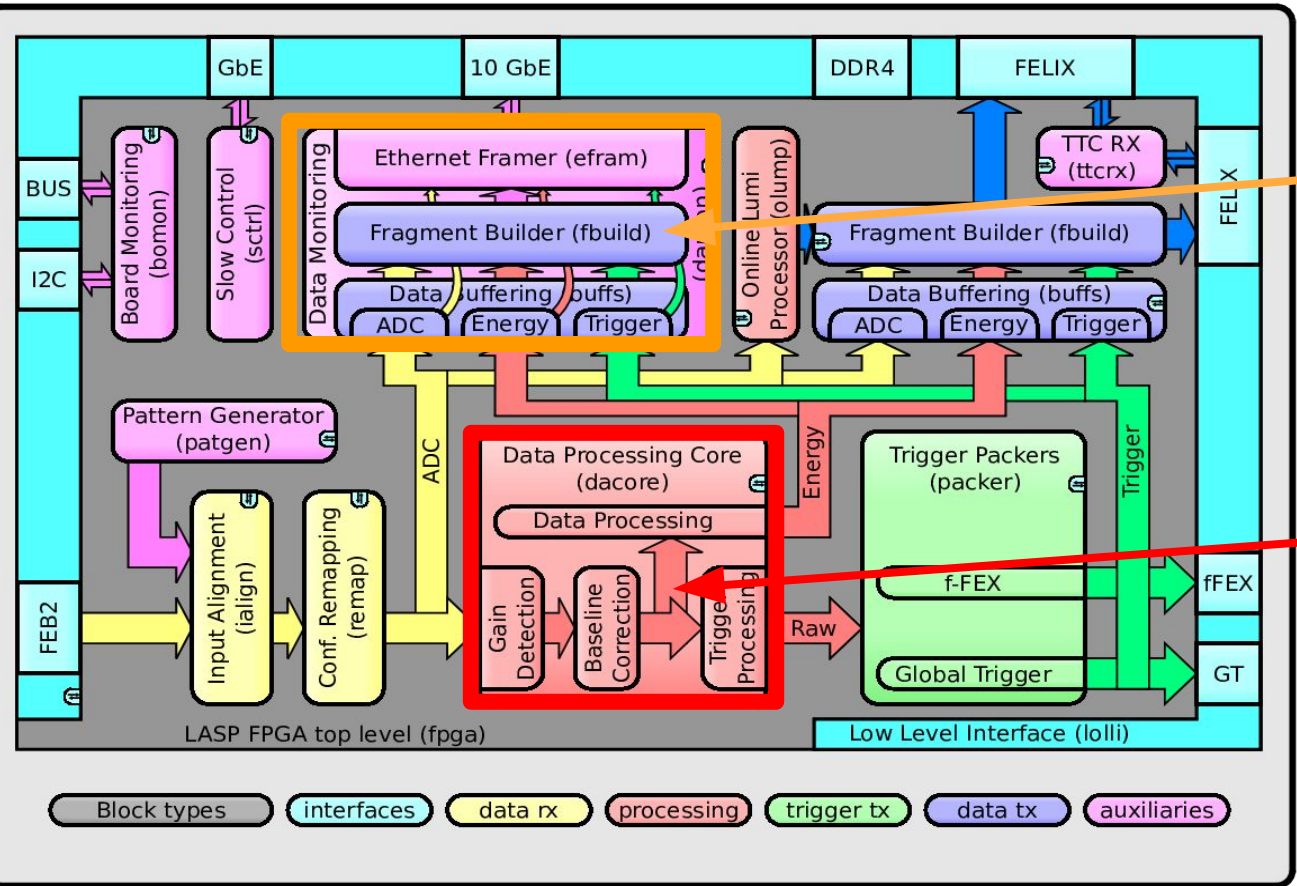
LAr Upgrade

- Full electronics of the readout path will be exchanged
 - New on-detector electronics that will digitize the signal at 40 MHz and send it to the backend
 - New off-detector electronics to compute the energy at 40 MHz



LASP Firmware

- LASP board containing 2 processing units based on INTEL FPGAs
 - Demonstrator board available with stratix 10 FPGAs
 - baseline for the firmware development shown in this talk
 - Final board will be equipped with Agilex FPGAs
- One FPGA should process **384 channels**
 - About **125 ns** allocated latency for energy computation

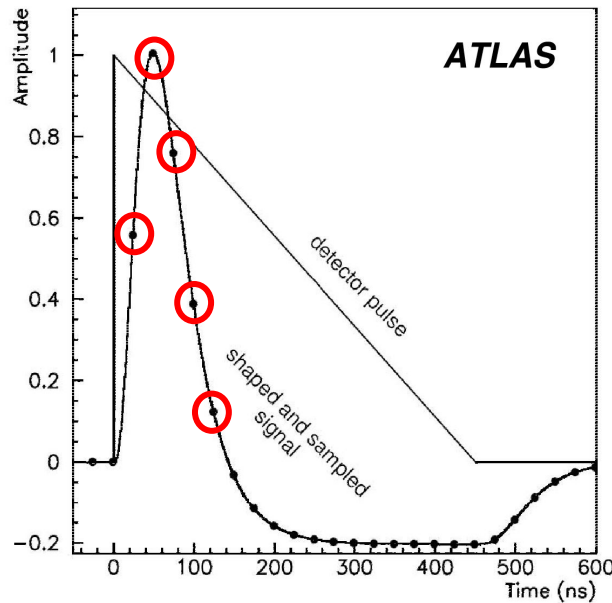


Buffering waiting for L1A
 Could be used to refine energy computation with less stringent latency requirements

Compute energy at 40 MHz
 Assign the energy to the correct bunch crossing (collision time)

Energy reconstruction

- Energy reconstruction currently using an optimal filtering algorithm with max finder
 - Optimal filtering to reconstruct the pulse and determine its amplitude (\propto energy)
 - Max finder to determine the correct time (bunch crossing)
- Not robust in case of distorted shapes due to pileup



Energy from Optimal-Filter (OF)

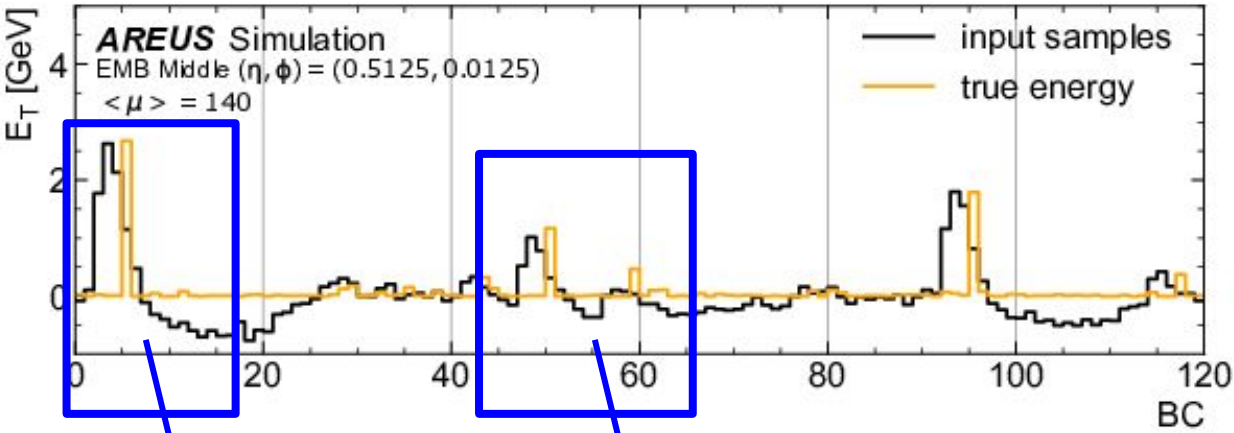
$n = 5$ in this talk

$$E(t) = \sum_{i=t}^{t+n} a_i \cdot s_i$$

Pre-set coefficients (fit of the peak)

Pulse Samples

Detailed description: The equation shows the energy E(t) as a sum of pre-set coefficients a_i multiplied by pulse samples s_i, where the sum is taken from i=t to i=t+n. A green arrow points from s_i to 'Pulse Samples' and an orange arrow points from a_i to 'Pre-set coefficients (fit of the peak)'. The text 'n = 5 in this talk' is written above the equation.



Relatively isolated pulse

Overlapping pulses
Distorted pulse shape

Energy reconstruction with NNs

Two neural networks types tested:
Convolutional Neural Networks (CNNs)
and
Recursive Neural Networks (RNNs)

CNN Structure

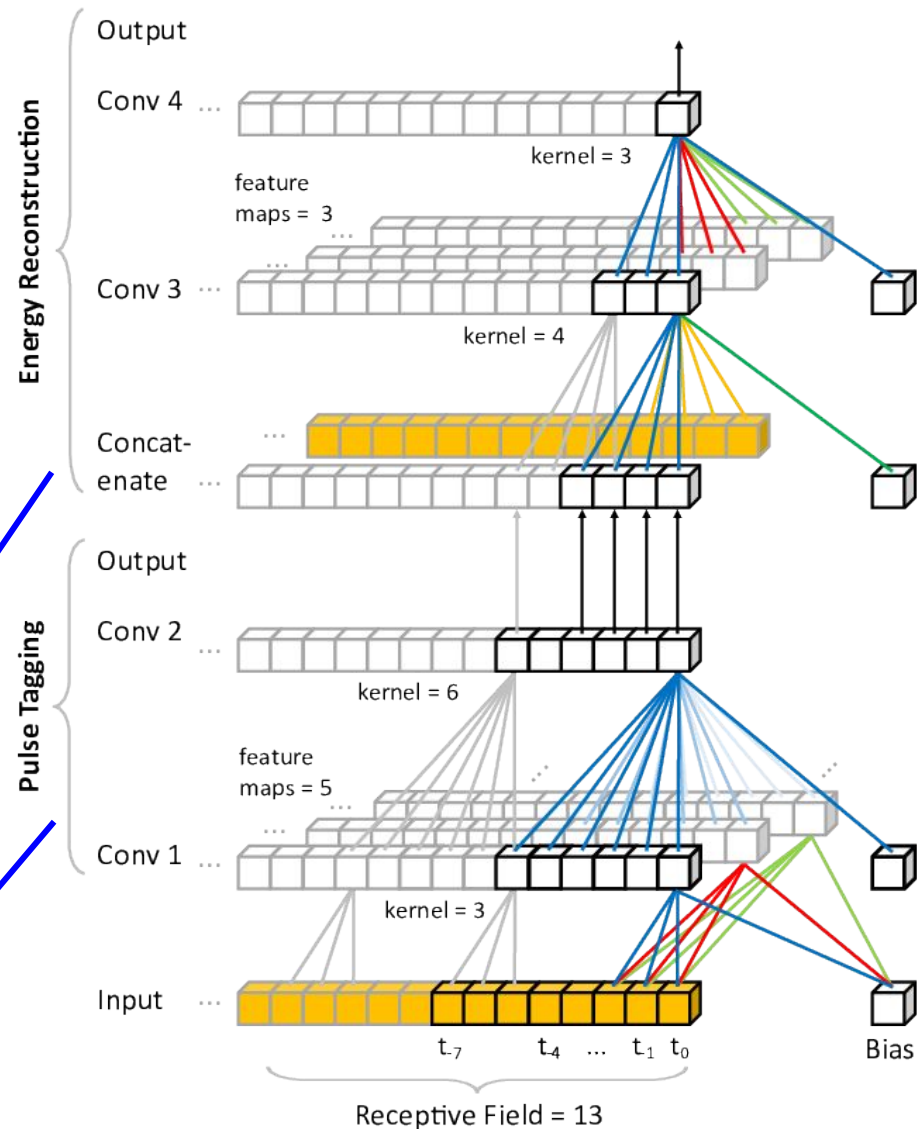
- Two blocks to identify the time of energy deposit and compute the corresponding energy
 - Two variations with 3 or 4 convolution layers (3-conv and 4-conv)
- 28 (13) samples as input for 3-conv (4-conv)
 - 23 (8) in the past (before the probed bunch crossing) to account for previous deposits

Energy computation block

Outputs the computed energy at a given bunch crossing

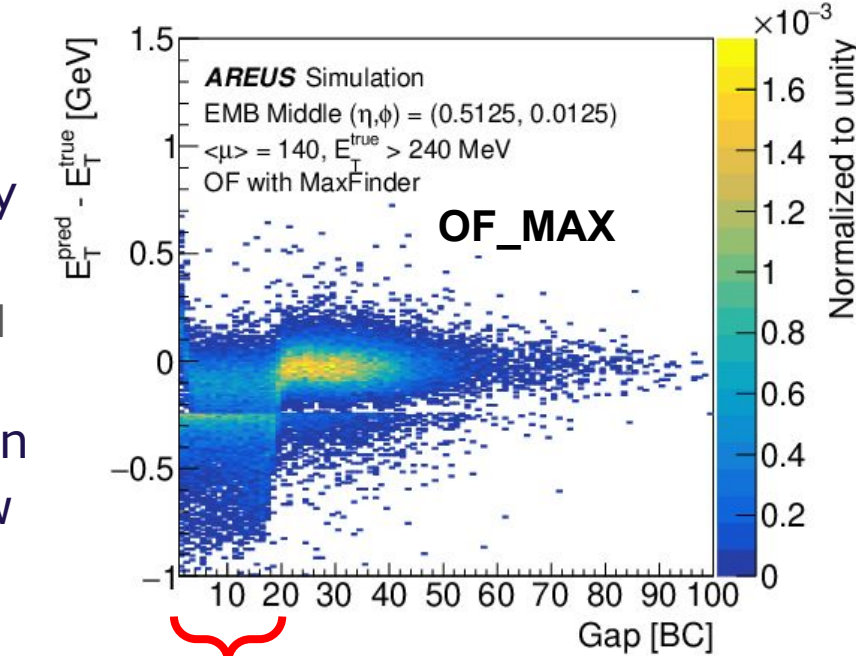
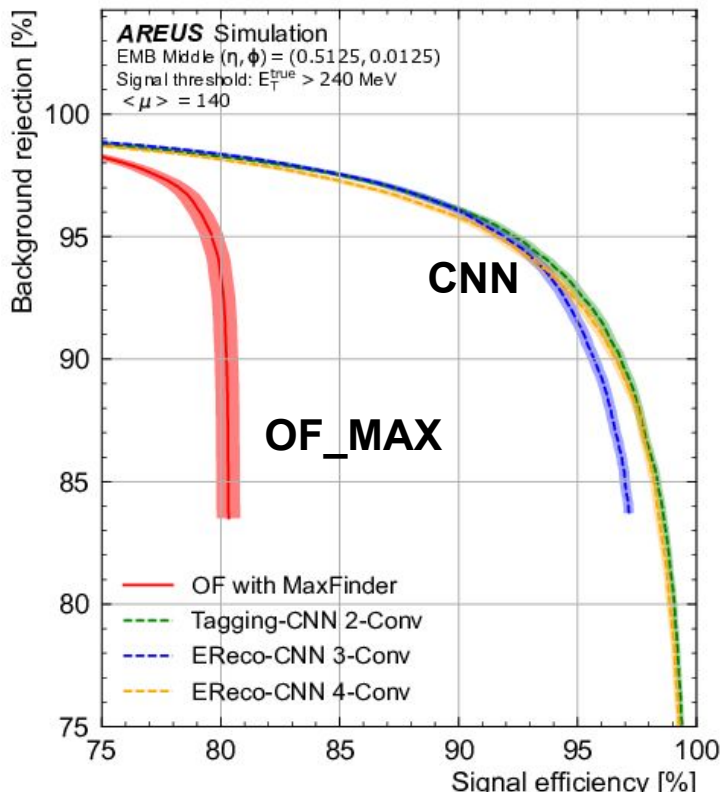
Time (bunch crossing) identification block

Outputs the probability of energy deposit (above noise) at a given bunch crossing

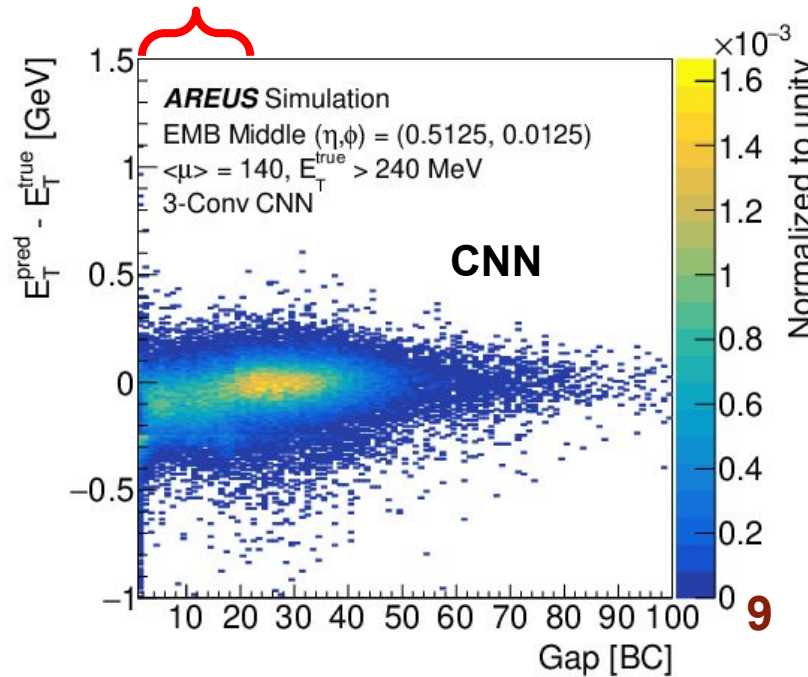


CNN Performance

- Improved efficiency in detecting energy deposits
 - Higher rejection of noise and mis-timed signals
- Clean improvement in energy resolution in the region where pulses overlap (low time gap between two signals)



Overlapping signals region

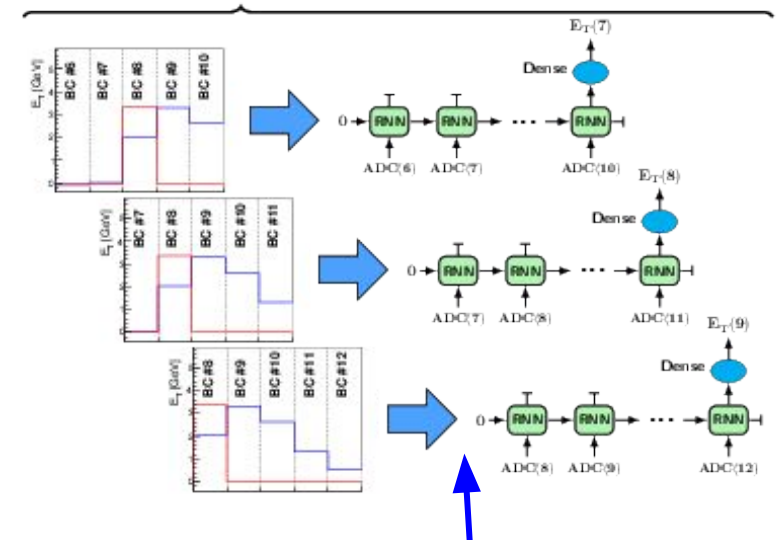
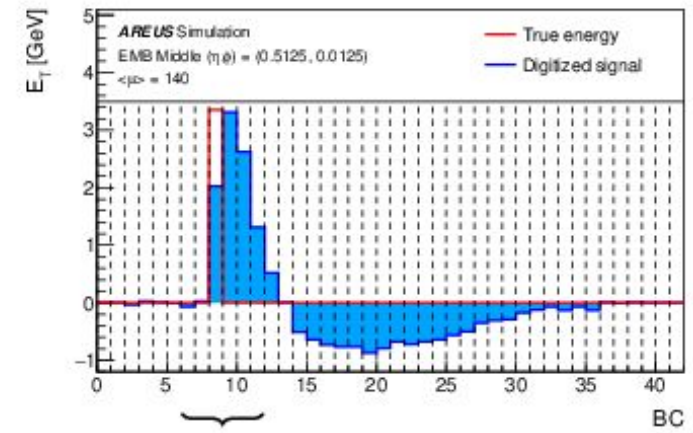
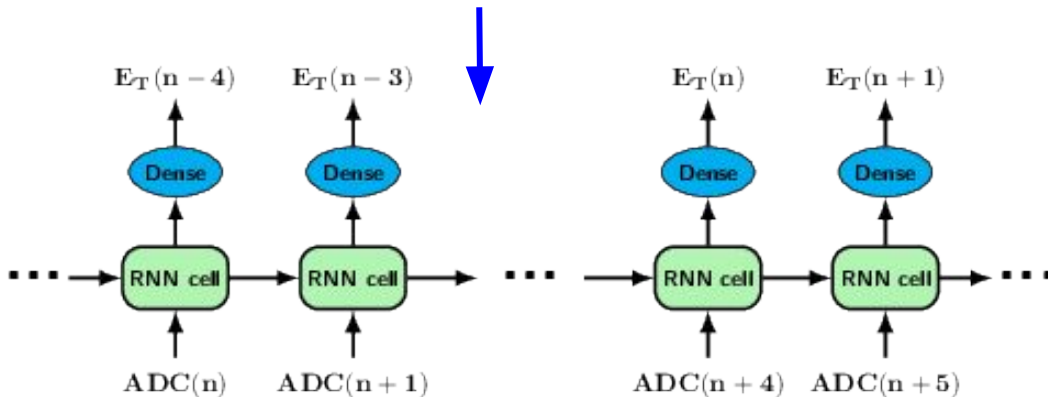


RNN structure

- Two architectures used
 - Single cell and Sliding windows
 - 4 samples corresponding to the signal are used
 - +1 in the past for the sliding windows
- Two types of RNNs
 - Vanilla RNN and LSTM

Single cell architecture

Continuous computation with a single cell
 Takes into account full past info (from the beginning of run)

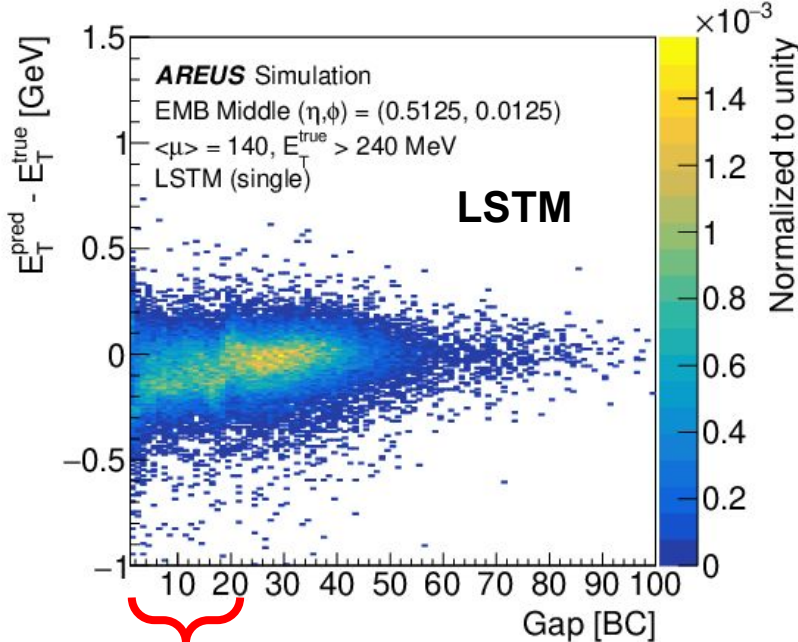


Sliding windows architecture

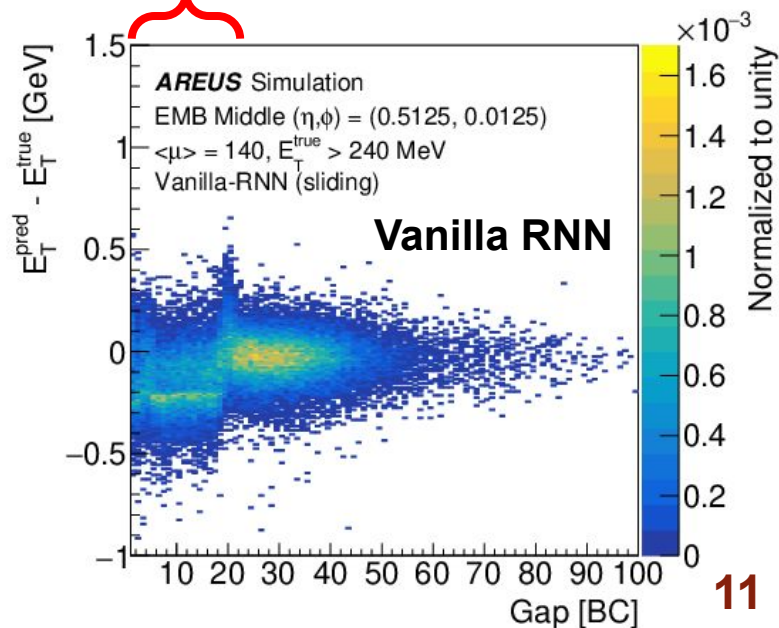
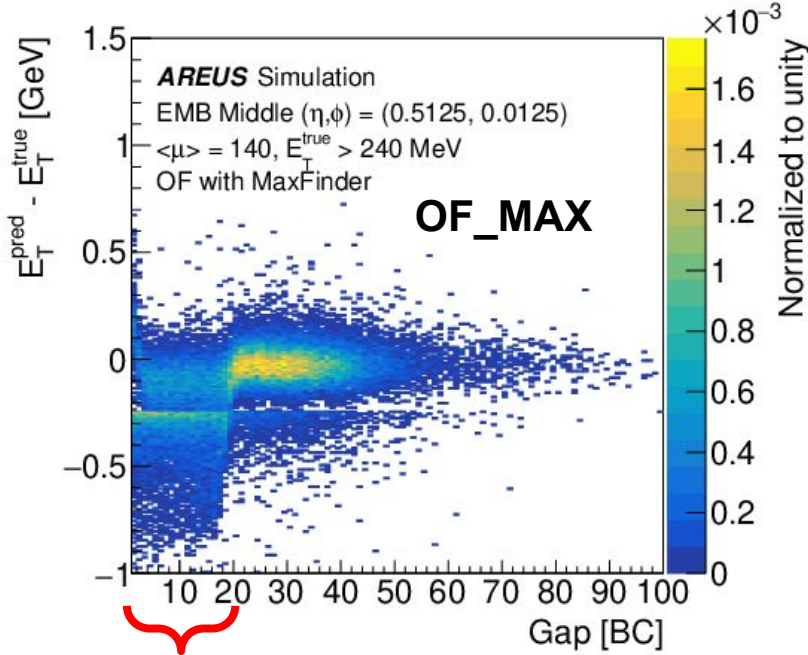
Computation on a moving slice of the data in fixed intervals
 Takes into account a limited set in the past (1 sample in the past in this talk)

RNN Performance

- LSTM with single cell recovers best the regions with low gap
 - Usage of full information from past events
- Vanilla RNN shows significant improvement with only 1 sample from the past

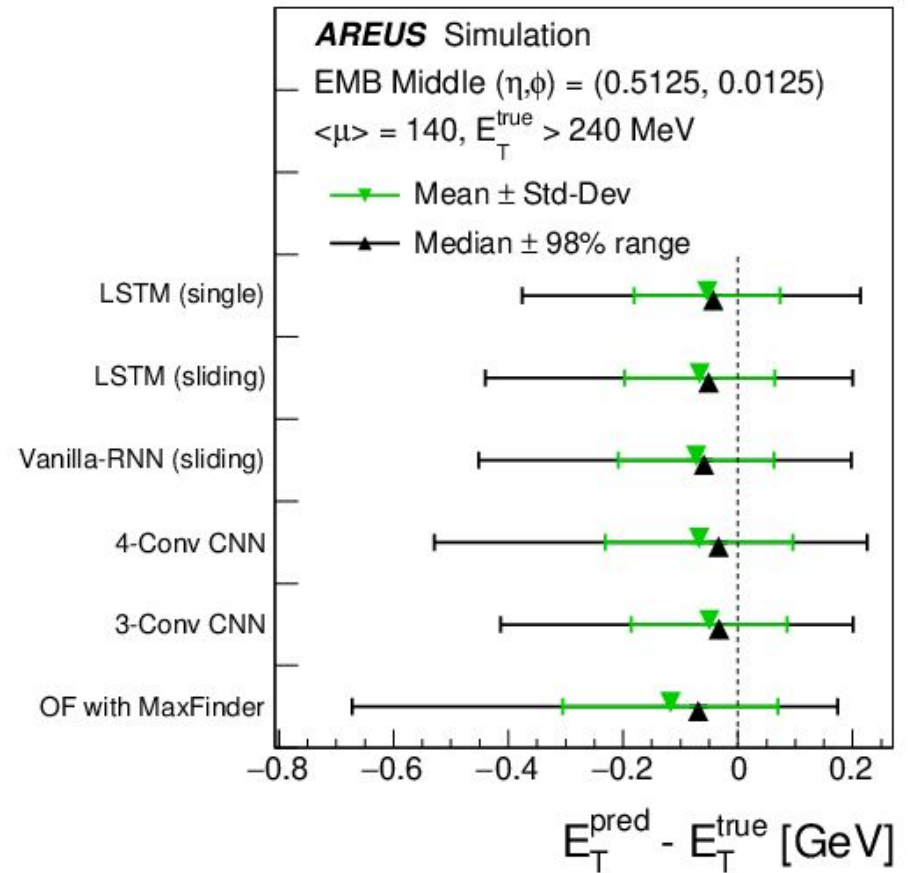


Overlapping signals region



NN Results

- Overall better energy scale and resolution for all NNs with respect to OF_MAX
 - Lower tails as seen from the 98% median range
- All NNs optimized to reduce as much as possible the required computing resources
 - While keeping good performance
- LSTM perform best but have a larger number of parameters
 - Harder to fit in the FPGAs



Algorithm	LSTM (single)	LSTM (sliding)	Vanilla (sliding)	CNN (3-conv)	CNN (4-conv)	Optimal filtering
Number of parameters	491	491	89	94	88	5
MAC units	480	2360	368	87	78	5

Firmware Implementation

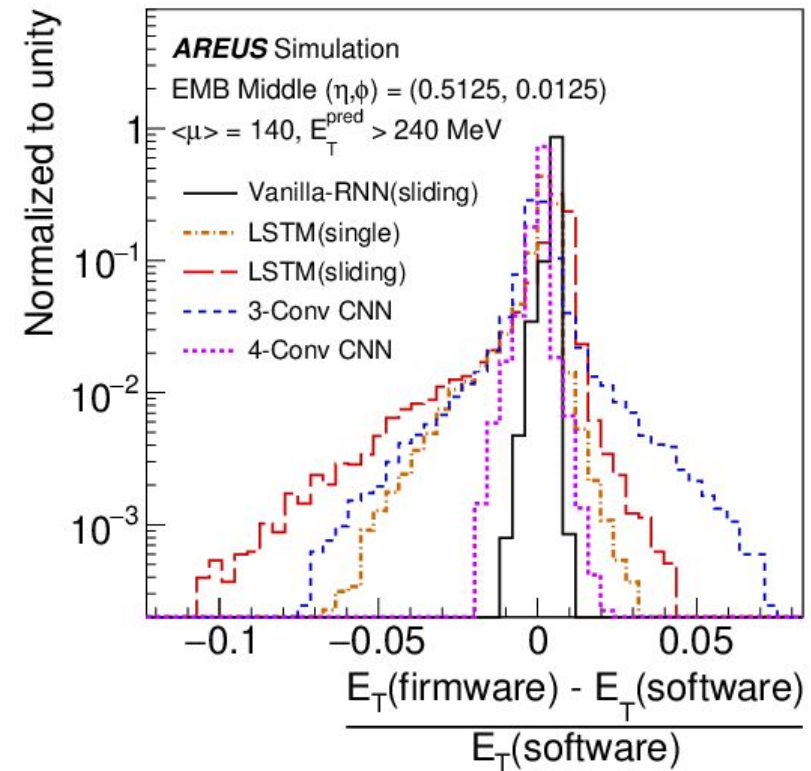
Focus on Vanilla RNN implementation in HLS and VHDL

384 channels per FPGA
125 ns latency

Firmware Implementation

- Implemented on Stratix 10 FPGA
 - reference 1SG280HU1F50E2VG
- CNN implemented in VHDL
- RNN implemented in HLS
- CNN and Vanilla RNN look feasible for the LAr usecase with time multiplexing

	3-Conv CNN	4-Conv CNN	Vanilla RNN (sliding)	LSTM (single)	LSTM (sliding)
Frequency F_{\max} [MHz]	493	480	641	560	517
Latency clk_{core} cycles	62	58	206	220	363
Initiation interval clk_{core} cycles	1	1	1	220	1
Resource Usage					
#DSPs	46 0.8%	42 0.7%	34 0.6%	176 3.1%	738 12.8%
#ALMs	5684 0.6%	5702 0.6%	13115 1.4%	18079 1.9%	69892 7.5%



- Energy computed with quartus simulation and verified on target
- O(1%) resolution due to firmware approximations
 - LUT for activation functions
 - Fixed point arithmetics

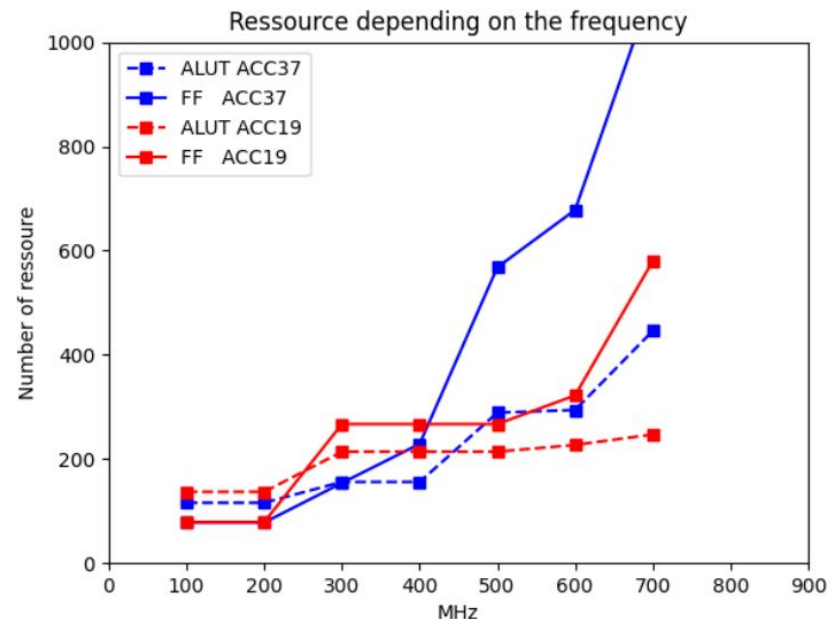
HLS optimisations (Vanilla RNN)

- Optimisation needed to fit vanilla RNNs that can compute 384 channels within 125 ns
 - Multiplexed RNNs use more resources as we add more networks to the FPGA
- Several optimisations are performed
 - Activation functions in LUT
 - Number of bits in fixed point representation
 - Rounding and truncation in arithmetic operations
 - Implementation of vector/matrix multiplication (Dot product)
- Dot product implementations
 - Naive C++: let HLS do it all
 - ACC37: accumulate (sum) in DSPs by chaining them
 - ACC19: ACC in ALUT

$$A.B = \begin{bmatrix} a_0 \\ a_1 \\ \vdots \\ a_8 \end{bmatrix} \cdot \begin{bmatrix} b_0 \\ b_1 \\ \vdots \\ b_8 \end{bmatrix} = \sum_{i=0}^7 a_i \cdot b_i$$

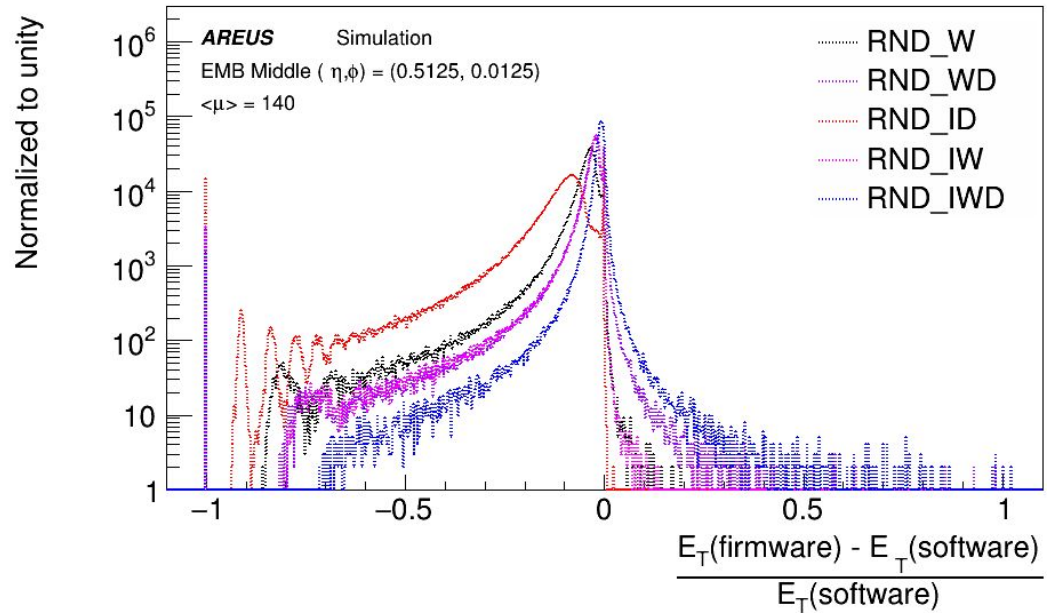
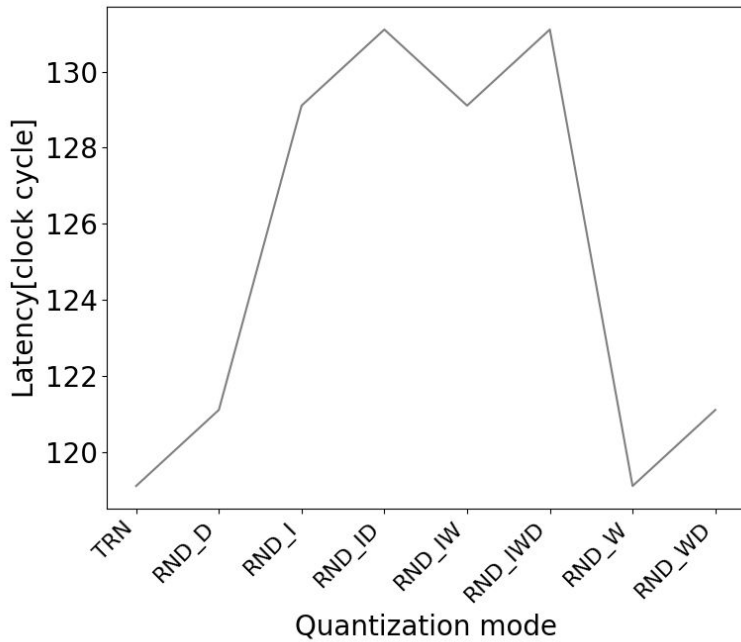
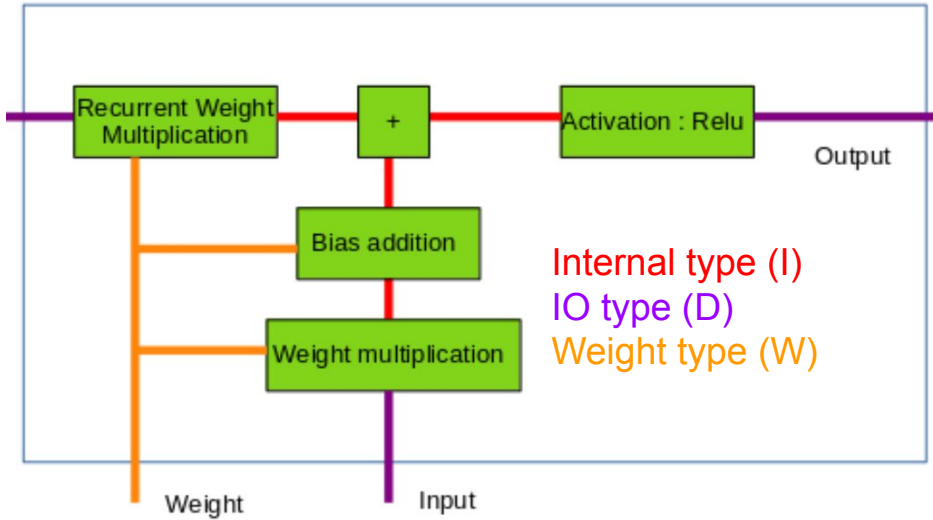
Implementation	ALUTs	FF	DSP
C++ style	709	222	8
ACC37	116	79	4
ACC19	137	78	4

@100 MHz



Rounding (Vanilla RNN)

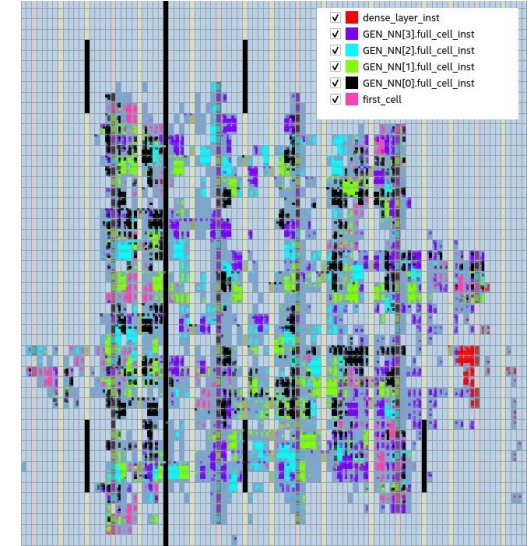
- Compromise between energy resolution and resource usage and latency
 - Truncation of IO and Internal types leads to important reduction of latency with small impact on energy resolution
 - Weight type rounded in software
 - No impact on latency



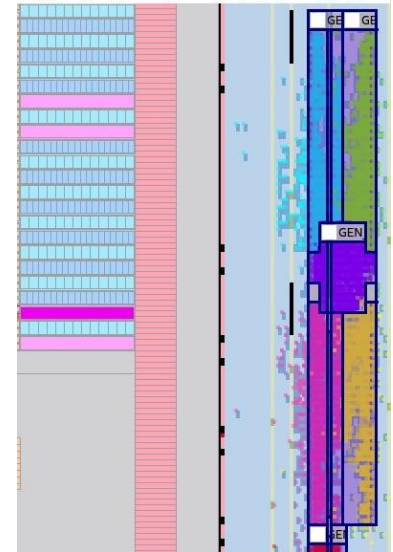
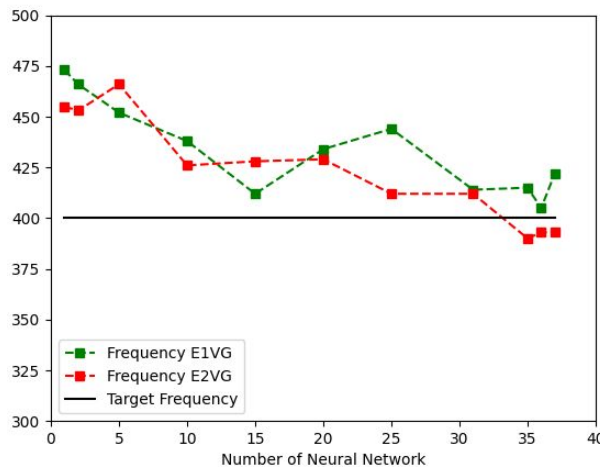
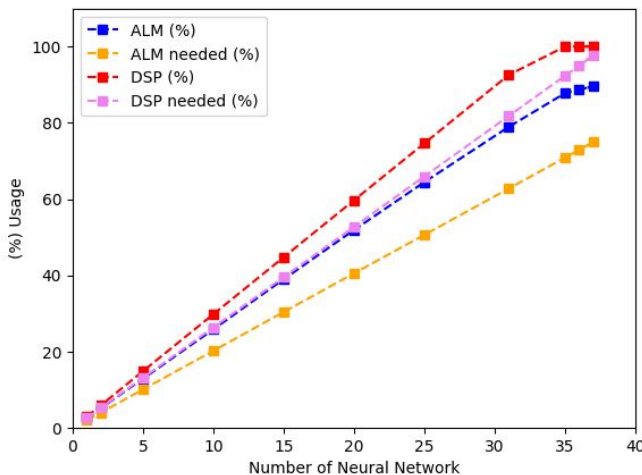
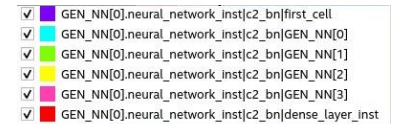
VHDL implementation of Vanilla RNN

- HLS does not allow to reach the target frequency and resource usage
 - Increase of the RNN ALM resources and reduction of FMax as we add networks to the FPGA
- Move to VHDL for the final fine tuning
- Force placement of the RNN components
 - Allow to better tackle timing violations and improve FMax
- Use incremental compilation
 - Keep networks with no timing violations and recompile only the rest

HLS placement



VHDL forced placement



RNN firmware results

- HLS allows fast development and optimisation of the firmware
 - Multiple developments and optimisations of RNN firmware in a short time
 - RNN for INTEL FPGAs implemented in [HLS4ML](#) for wider usage
- VHDL is needed to fine tune the design and fit the LAr requirements
- Vanilla RNN firmware produced and fit the requirements with Stratix 10
 - Better performance expected with the Agilex FPGA
 - However still need to test it within the full LASP firmware

	N networks x multiplexing	ALM	DSP	FMax	latency
target	384 channels	30%*	70%*	-	125 ns
HLS (no multiplexing)	384x1	226%	529%	-	322 ns
HLS optimized	37x10	23%	100%	414 MHz	302 ns
VHDL optimized	28x14	18%	66%	561 MHz	121 ns

*based on experience with the phase I upgrade

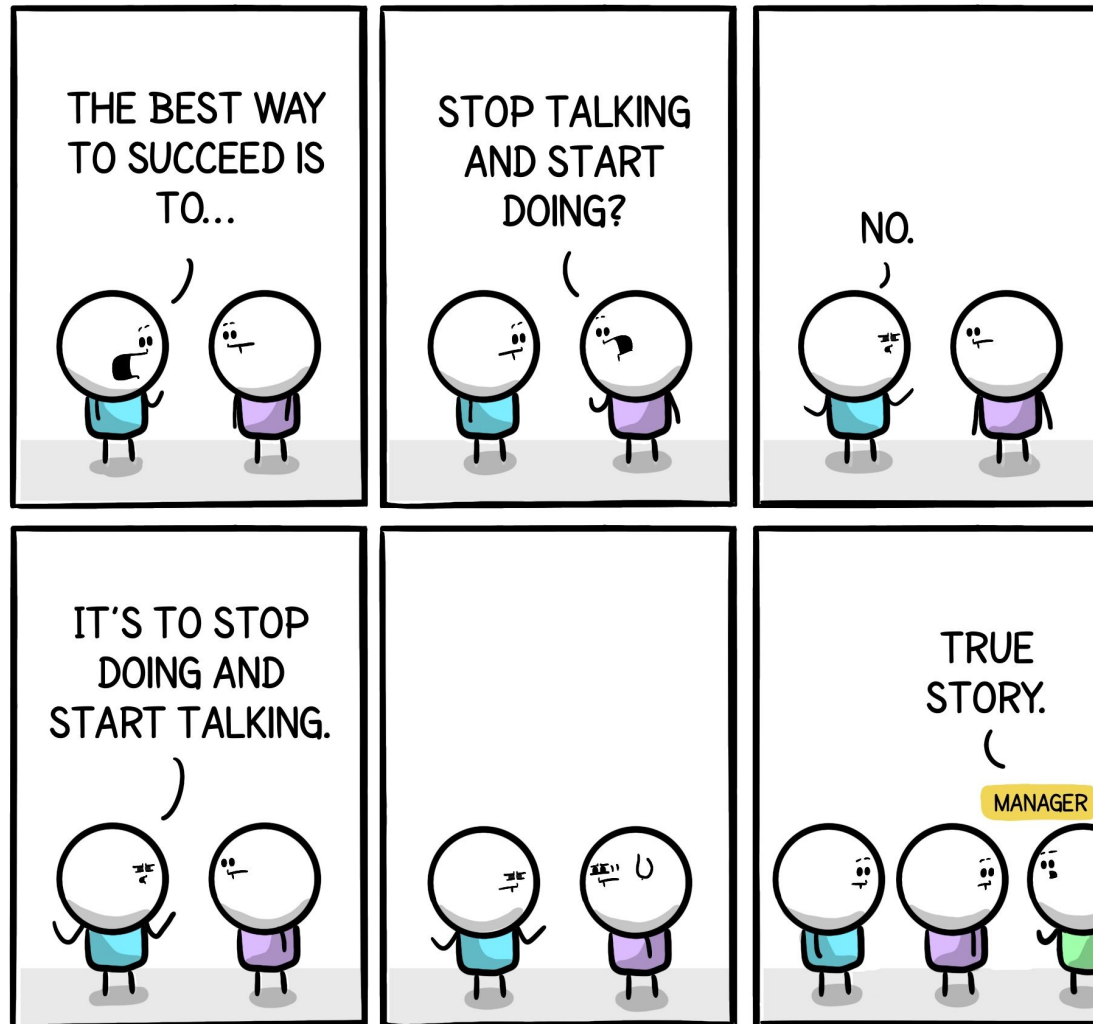
Conclusion

- CNN and RNN networks outperform the optimal filtering algorithm for the energy reconstruction in the ATLAS LAr Calorimeter
 - Particularly in the region with overlap between multiple pulses
- Next step is to quantify the effect on object (electrons, photons) reconstruction and physics performance

- All networks are designed to reduce to a maximum the resource usage while keeping the performance
- CNN and Vanilla RNN are serious candidates that can fit the stringent requirements on the LASP firmware
- Vanilla RNN and CNN optimisations allow to reach the requirements in terms of resource usage and latency
 - Testing on hardware (INTEL DevKits) started and shows good results
 - Need to check timing violations with all other components of the LASP firmware

Opening the way for AI usage to process raw detector data online and
at trigger level

Backup

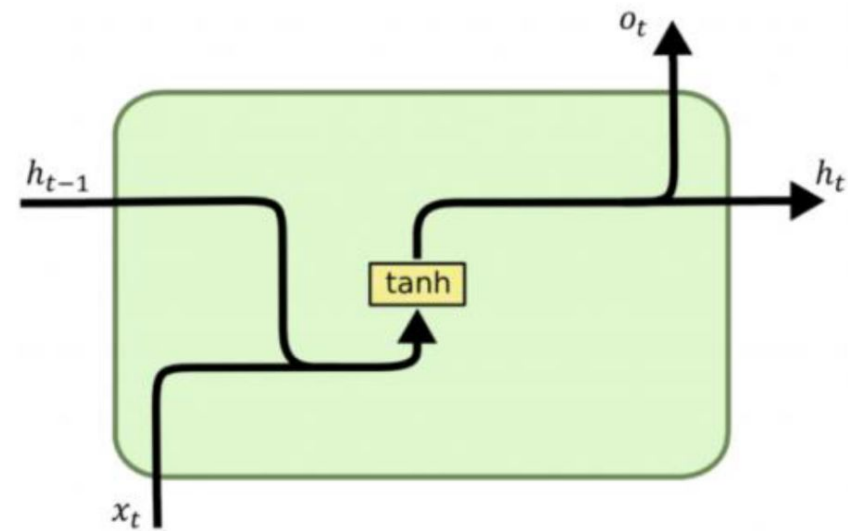


Hello. I make comics about work.
Follow me on Instagram / Twitter / Facebook.

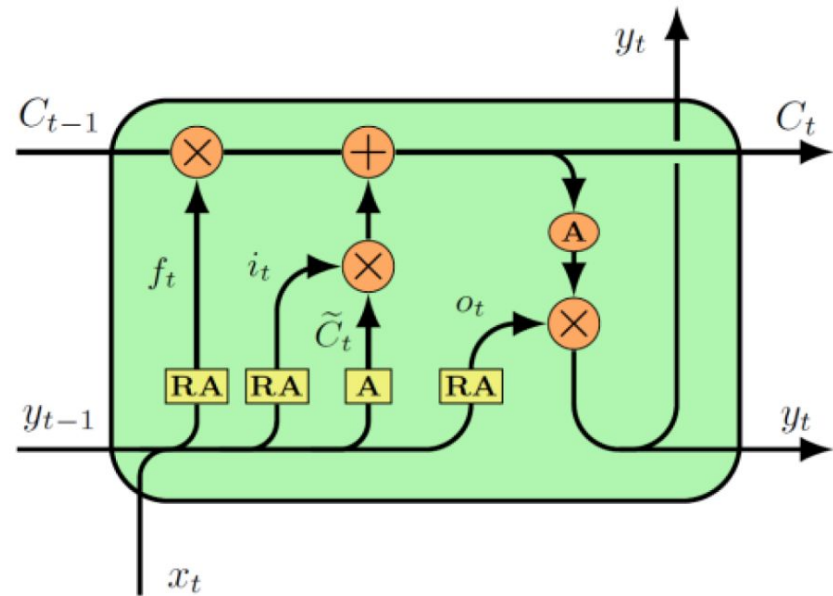
Work Chronicles
workchronicles.com

RNN cells

Vanilla RNN cell structure



LSTM cell structure



CNN configuration

Table 1 CNN configurations with one and two energy reconstruction layers and identical tagging layers.

	3-Conv			4-Conv			
	Tagging		Energy Re- construction	Tagging		Energy Re- construction	
Layer index	1	2	3	1	2	3	4
Kernel Size	3	6	21	3	6	4	3
Dilation Rate	1	1	1	1	1	1	1
Feature Maps	5	1	1	5	1	3	1
Activation Function	sigmoid		ReLU	sigmoid		ReLU	
Number of Parameters	51		43	51		37	
MAC units	45		42	45		33	
Receptive Field	28			13			

RNN configuration

Table 2 Configurable key parameters of the single-cell and sliding-window algorithms.

		Single-cell LSTM	Sliding-window	
			LSTM	Vanilla RNN
Time inference	Receptive Field	∞	5	5
	Samples after deposit	5	4	4
RNN layer	Dimension	10	10	8
	Activation	tanh	tanh	ReLU
	Recurrent Activation	sigmoid	sigmoid	N/A
Dense layer	Dimension	1	1	1
	Activation	ReLU	ReLU	ReLU
Number of Parameters		491	491	89
MAC units		480	2360	368

Firmware properties for a single network

Table 3 Performance of the VHDL implementation of CNNs and the HLS implementation of RNNs compiled with Quartus 20.4 [13] for a Stratix-10 FPGA (reference 1SG280HU1F50E2VG) and single data input channel.

	3-Conv CNN	4-Conv CNN	Vanilla RNN (sliding)	LSTM (single)	LSTM (sliding)
Frequency F_{\max} [MHz]	493	480	641	560	517
Latency clk_{core} cycles	62	58	206	220	363
Initiation interval clk_{core} cycles	1	1	1	220	1
Resource Usage					
#DSPs	46 0.8%	42 0.7%	34 0.6%	176 3.1%	738 12.8%
#ALMs	5684 0.6%	5702 0.6%	13115 1.4%	18079 1.9%	69892 7.5%

Firmware properties for multiplexed networks

Table 4 Multiplexing performance of the VHDL implementation of CNNs and the HLS implementation of RNNs compiled with Quartus 20.4 [13] for a Stratix-10 FPGA (reference 1SG280HU1F50E2VG).

	3-Conv CNN	4-Conv CNN	Vanilla RNN
Multiplicity	6	6	15
Frequency F_{\max} [MHz]	344	334	640
Latency clk_{core} cycles	81	62	120
Initiation interval clk_{core} cycles	1	1	1
Max. Channels	390	352	576
Resource Usage			
#DSPs	46 0.8%	42 0.7%	152 2.6%
#ALMs	14235 1.5%	15627 1.7%	5782 0.6%

Dot product implementations

Naive C++ implementation

```
for (int i=0; i < 8; i++){  
    acc += a[i] * b[i];  
}
```

ACC37 implementation

```
for (int i=0; i < 4; i++){  
    tmp[i] = a[i]*b[i] + a[7-i]*b[7-i];  
}  
for (int i=0; i < 4; i++){  
    acc += tmp[i];  
}
```

ACC19 implementation

```
for (int i=0; i < 4; i++){  
    tmp[i] = hls_fpga_reg(a[i]*b[i] + a[7-i]*b[7-i]);  
}  
for (int i=0; i < 4; i++){  
    acc += tmp[i];  
}
```

DSP

