Machine Learning for Real-Time Processing of ATLAS Liquid Argon Calorimeter Signals with FPGAs

November 19, 2021

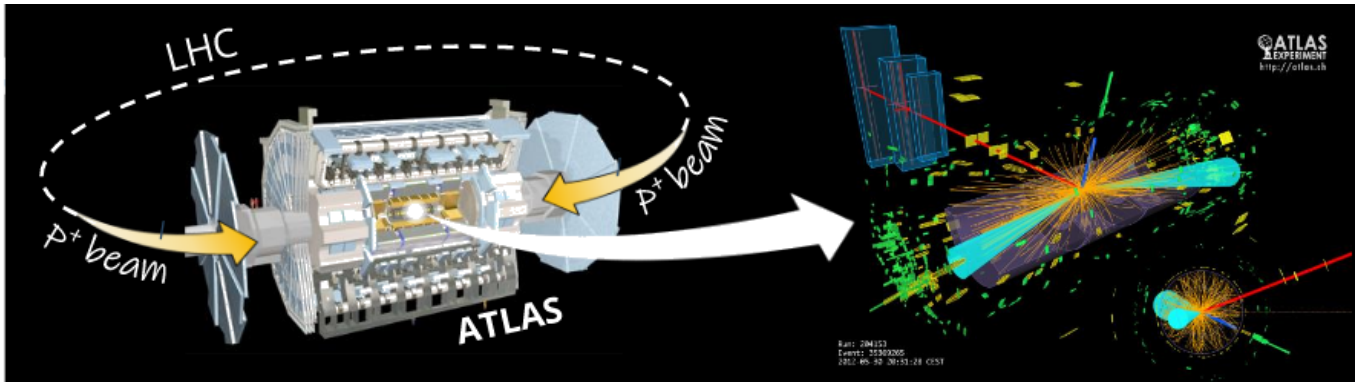
Etienne FORTIN

CPPM and Dresden LAr group

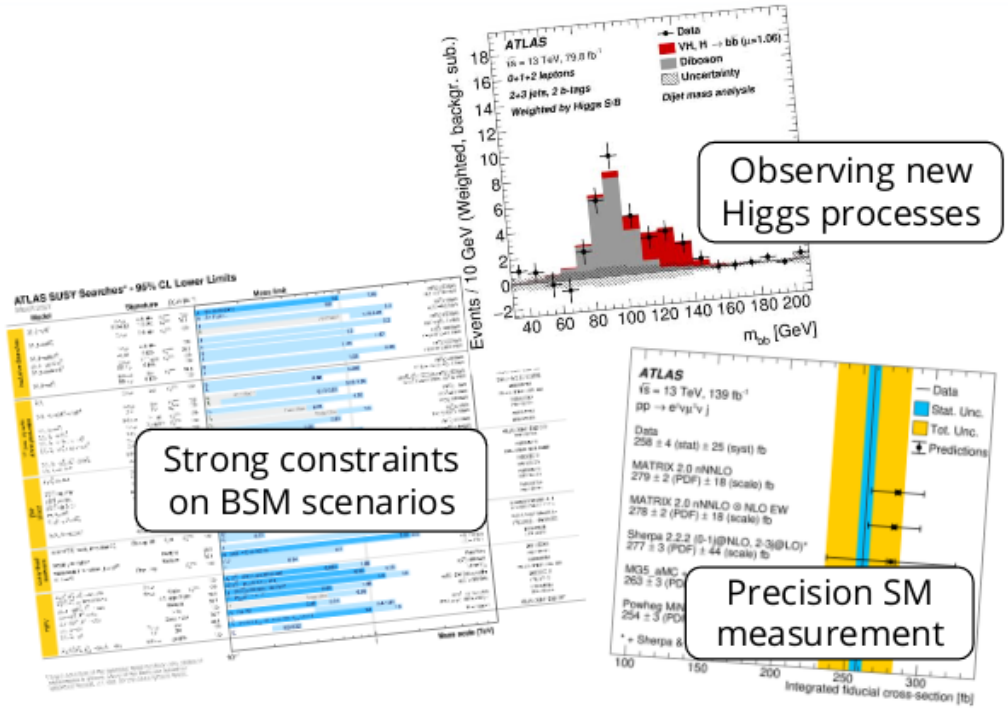
Overview

ATLAS

ATLAS :
a general purpose detector at LHC

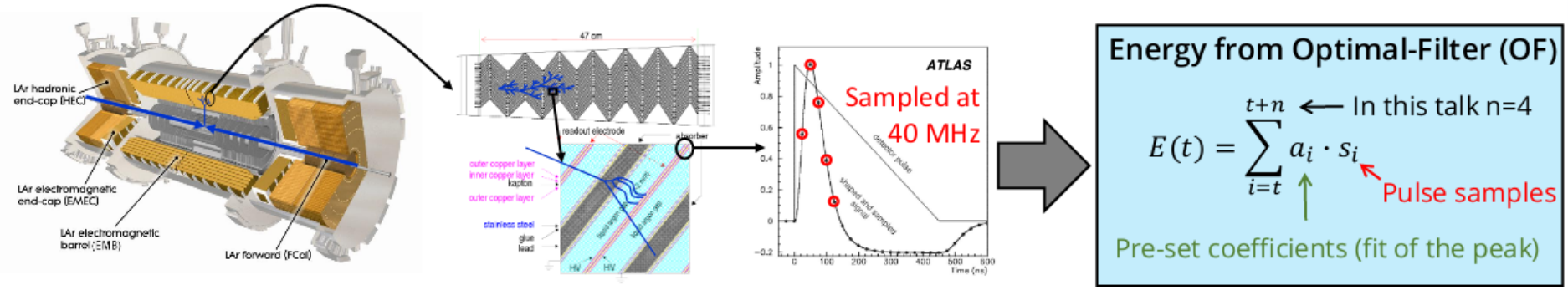


- Efficient data collection
- High quality reconstruction for all p-p products

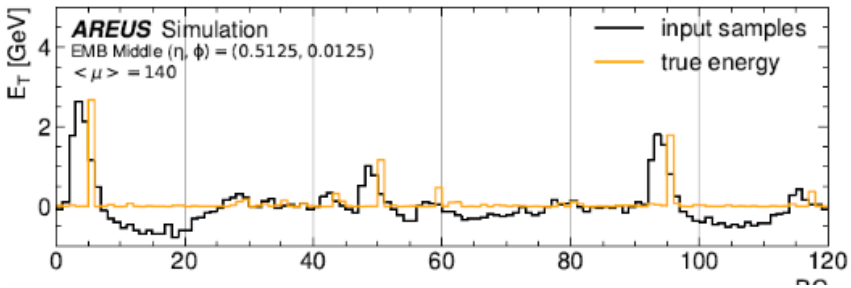


Liquid Argon Calorimeter

- Responsible for measuring energies of $e^{\pm}$, γ and forward hadrons (close to beam axis)



- In data taking conditions :

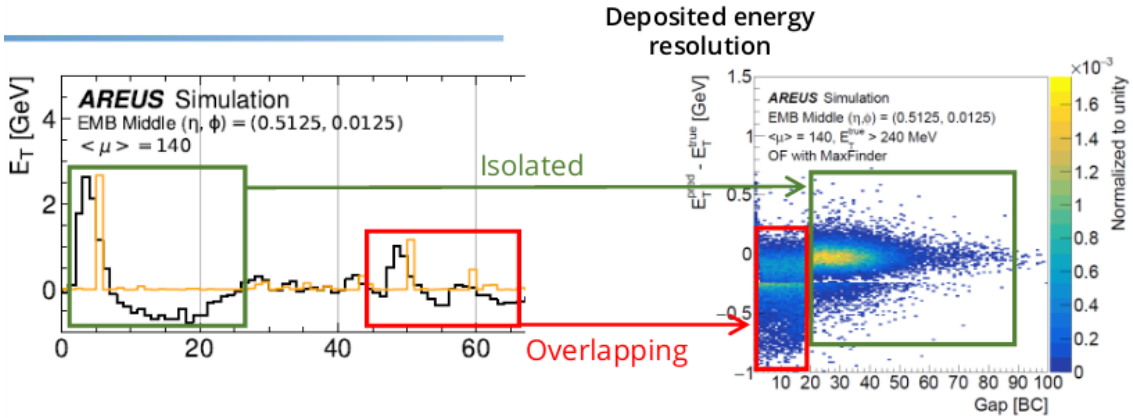


(1) **continuous** deposits at 40 MHz → require → Peak finder / trigger (When to apply OF)

(2) Real time energies for triggers → require → Compact algorithms on high end FPGA

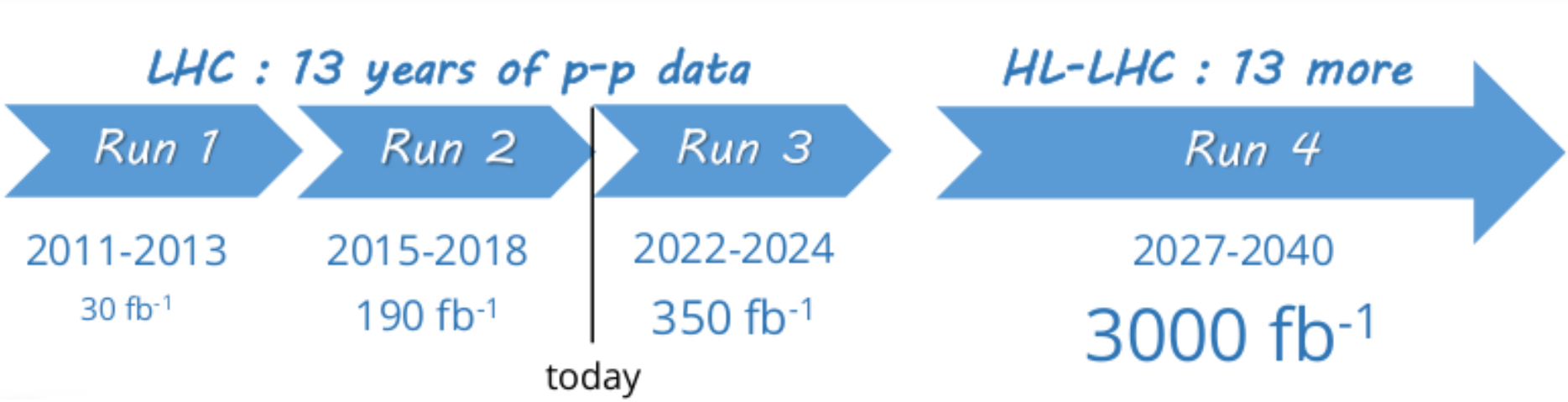
Liquid Argon Calorimeter

- OF algorithm cannot correct past events overlapping current event with small gap
- The Gap is the distance between two energy deposits

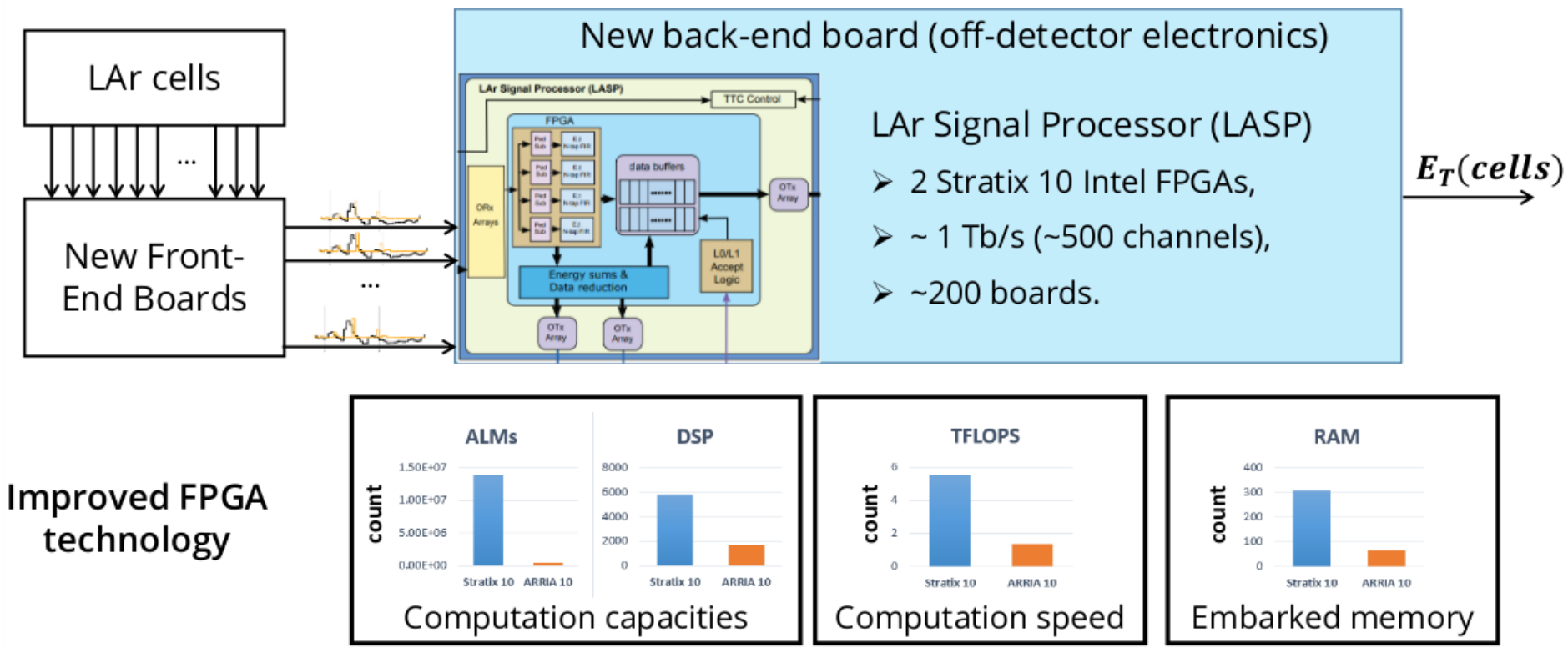


High Luminosity (HL-)LHC : crucial data for HEP analyses in the next two decades

- High pile-up (up to 200, now 30) will cause more overlapped and distorted pulse
- To keep the performance we need to improve the algorithms for energy computation



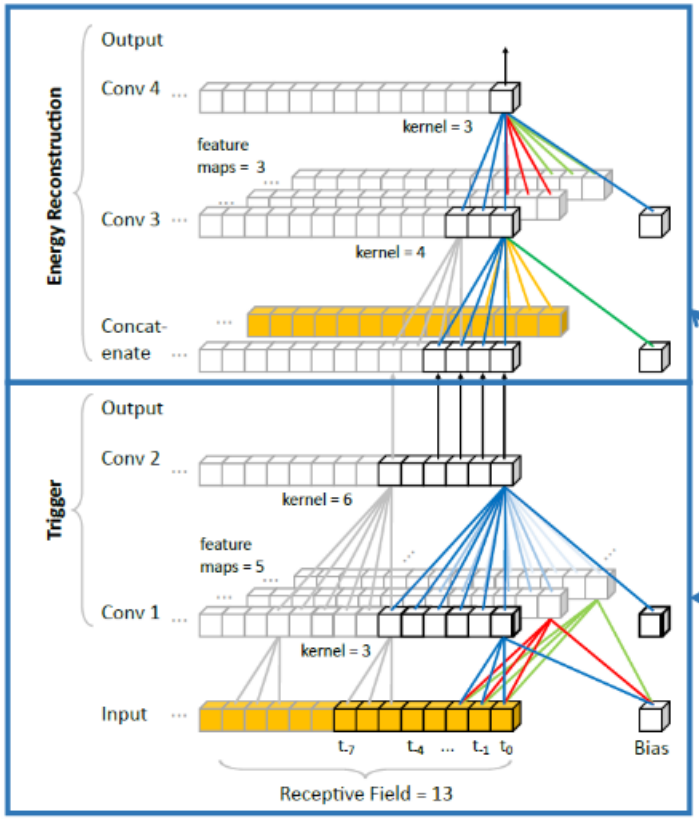
LAr Phase 2 upgrade

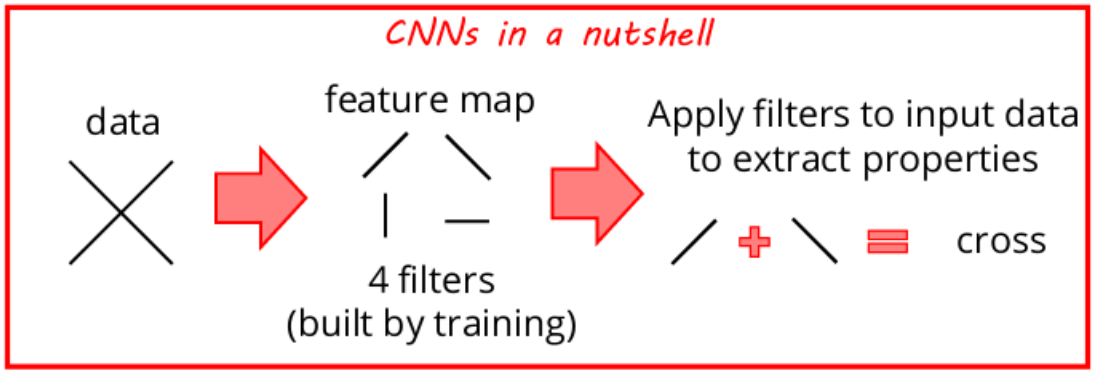


- The resources are now suffisant to implement Artificial Intelligence Algorithm in LAr Signal Processing boards or fpga

Energy inference using Convolutional Neural Networks

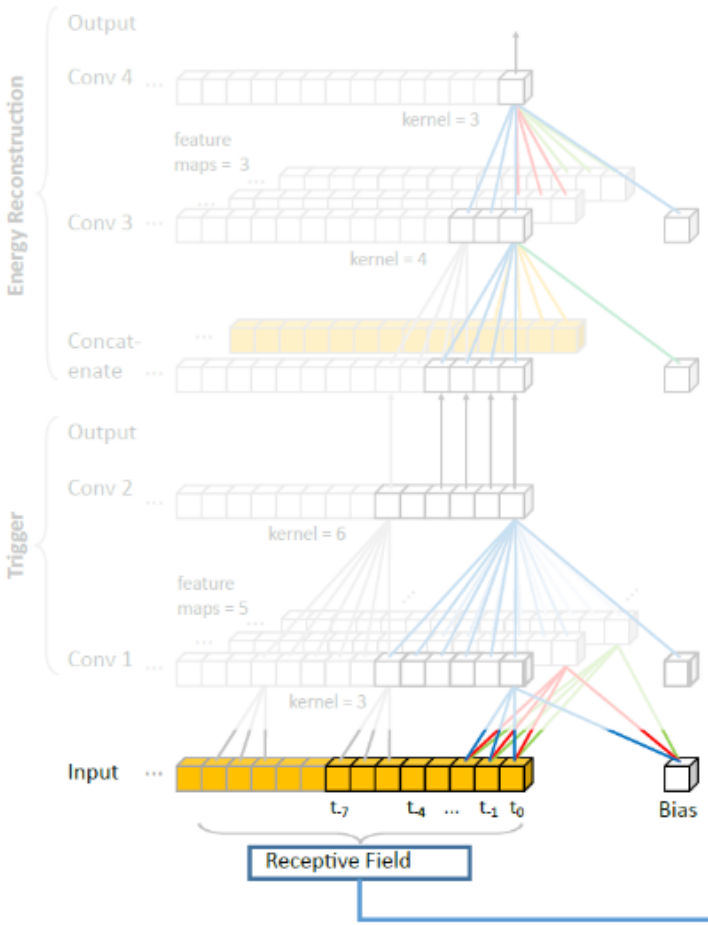
Convolutional Neural Netwwork: succession of filters (layers) extracting data properties

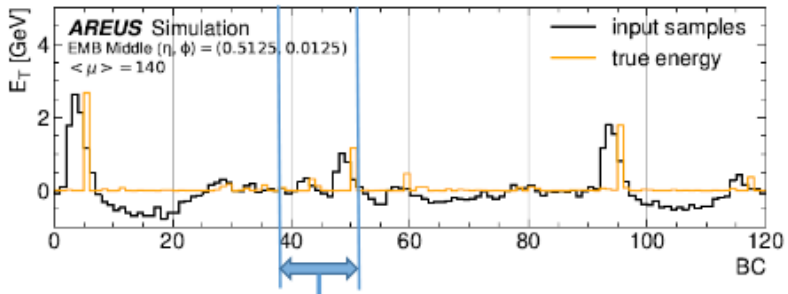




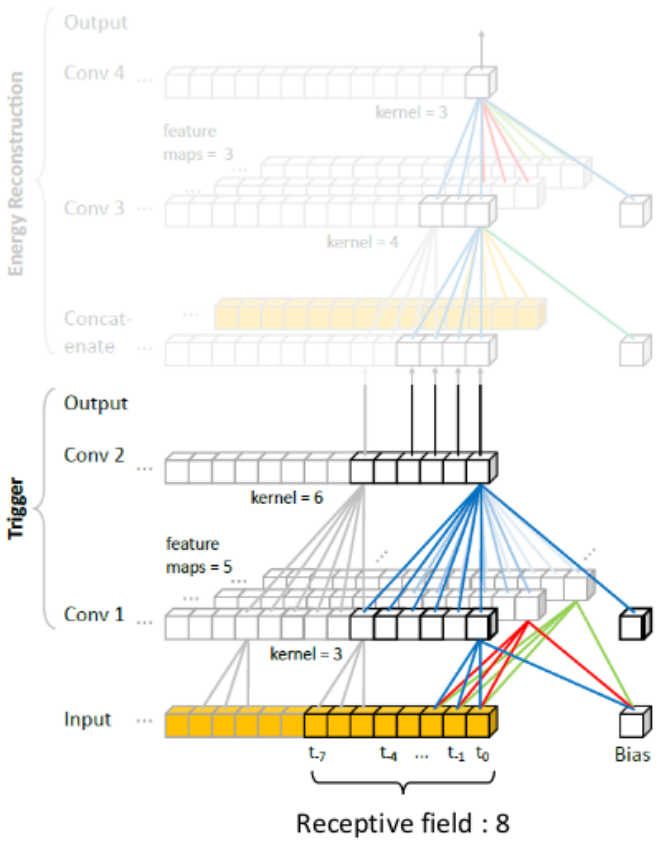
2 tasks : trigger and inference

CNN Inputs


Energy Reconstruction
Trigger
Output
Conv 4
kernel = 3
feature maps = 3
Conv 3
kernel = 4
Concat-enate
Output
Conv 2
kernel = 6
feature maps = 5
Conv 1
kernel = 3
Input
t_{-7} t_{-4} ... t_{-1} t_0
Bias
Receptive Field

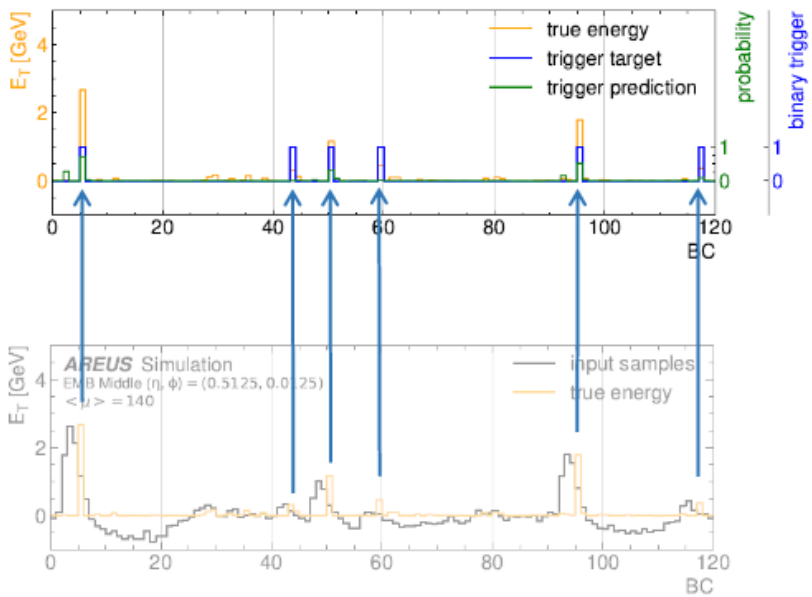

AREUS Simulation
EMB Middle (η, ϕ) = (0.5125, 0.0125)
< μ > = 140
input samples
true energy
E_T [GeV]
BC

Triggering with CNNs

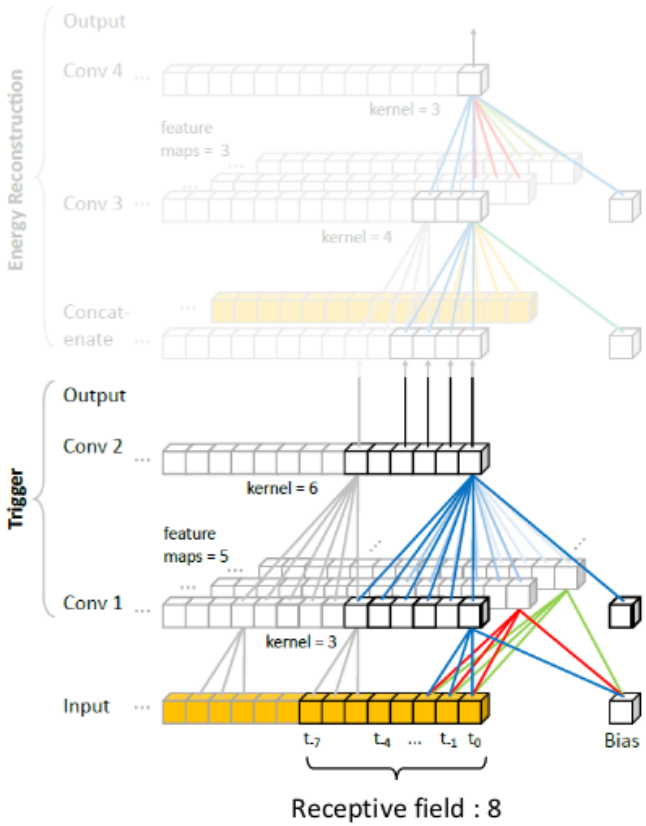


Trigger CNN :

trained to detect deposits 3σ above noise (240 MeV)
using pulse samples for 8 bunch-crossings

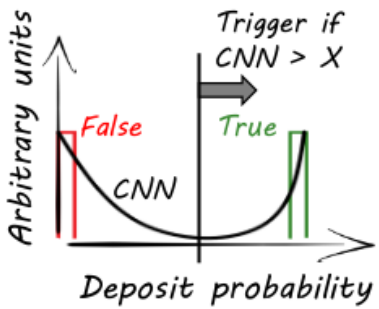


Triggering with CNNs

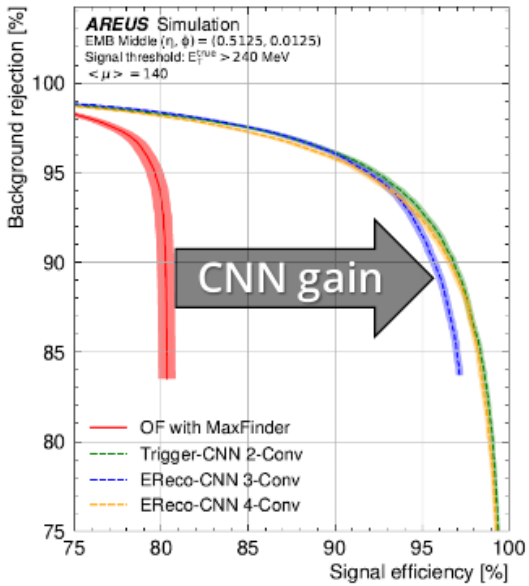


Trigger CNN :

trained to detect deposits 3σ above noise (240 MeV)
using pulse samples for 8 bunch-crossings



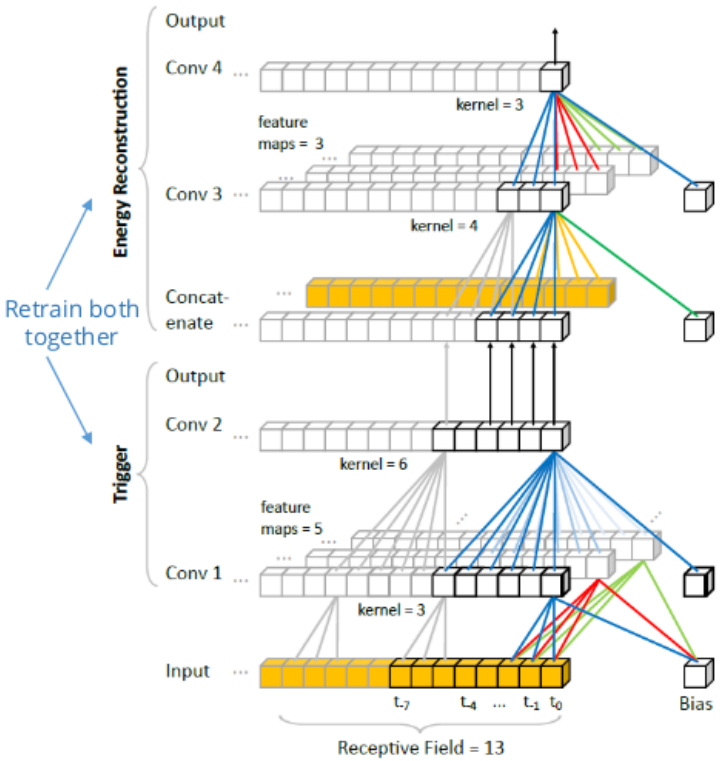
Efficiency to reject BC with deposits < 240 MeV

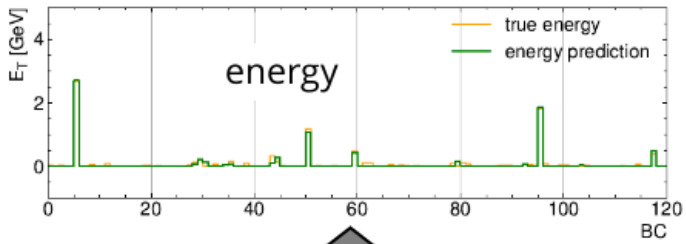


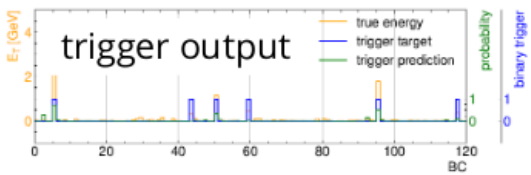
Efficiency to select BC with deposits > 240 MeV

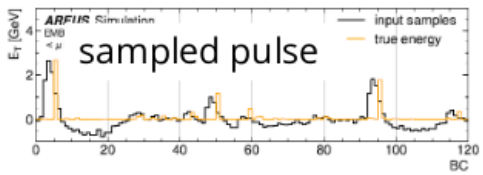
Energy inference using CNNs

Add CNN energy reconstruction layers









4-Conv CNN (left sketch)

2 layers in energy reconstruction

Receptive fields : 13 BC

=> 88 parameters total

3-Conv CNN

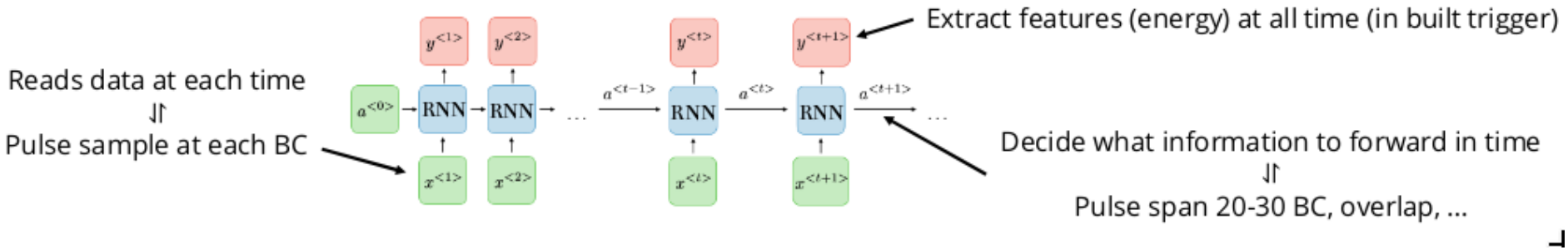
1 layer in energy reconstruction

Receptive fields : 28 BC

=> 94 parameters total

Recurrent Neural Network (RNN) architectures

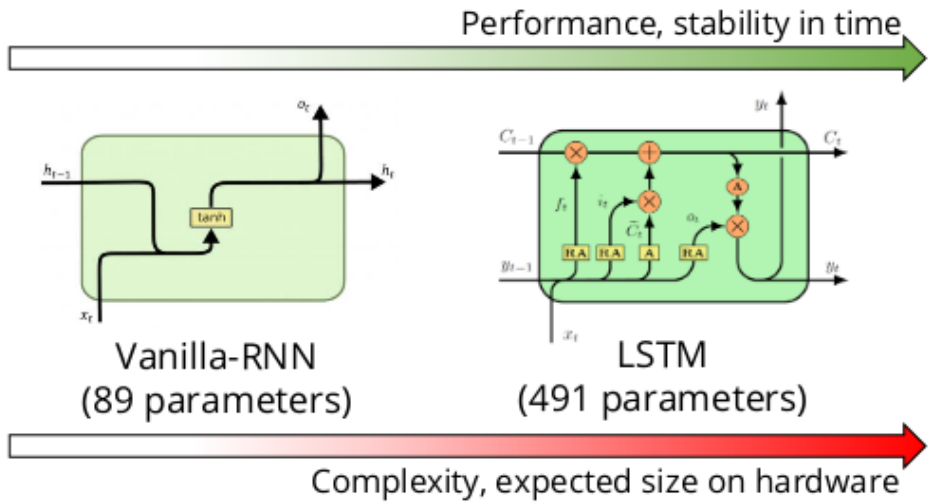
RNN : set of operations (internal NN) called upon arrival of each new data



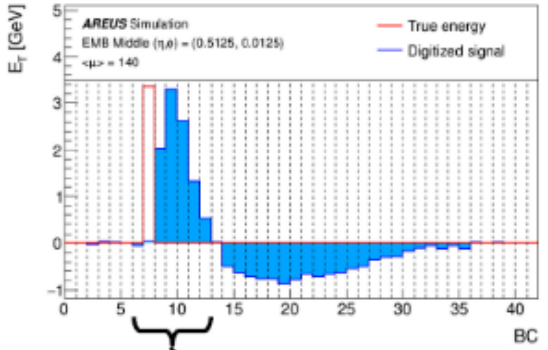
RNN are natural fit for inference of LAr energy deposits

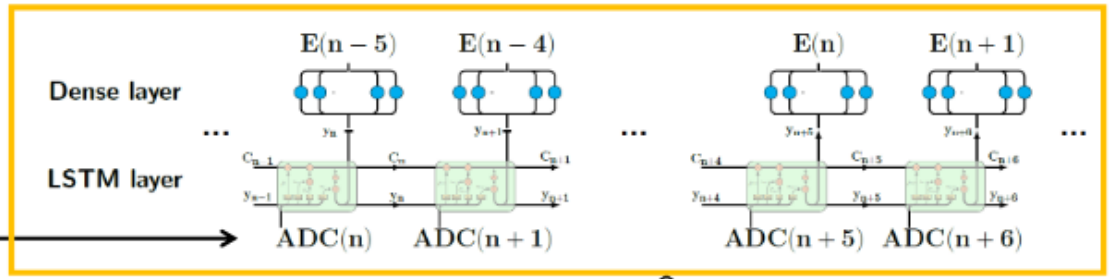
Two RNNs internal architectures explored

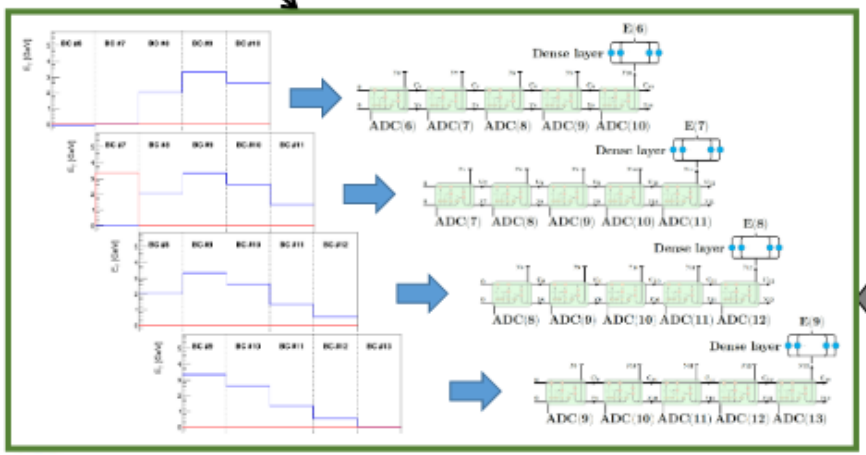
- Optimized for lower paramter count
- Long Short-Term Memory (LSTM) : 10 Internal dimensions
- Vanilla-RNN : 8 Internal dimensions



RNN applications to energy inference : 2 methods







Single cell method

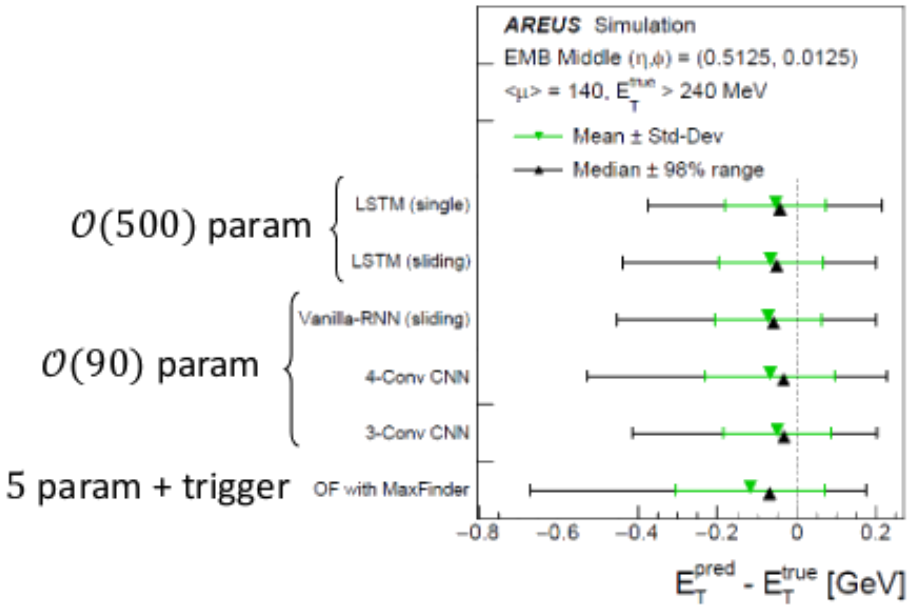
- ✓ Long range correction, standard RNN application.
- × Lots of logic to handle data in time (**only LSTM**).

Sliding window method (5 BC)

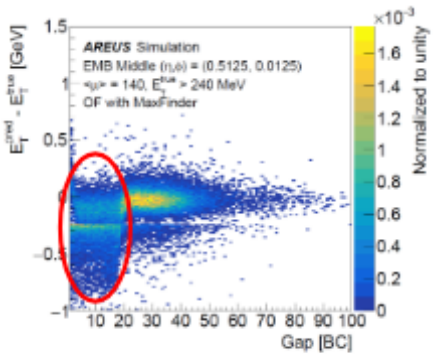
- ✓ Focus logic on inference, better stability.
- × Short range correction only (1 BC in the past).

Performance

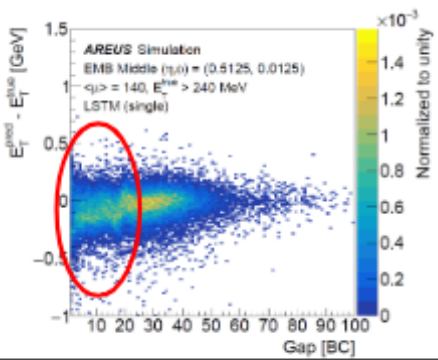
- Comparisons on single LAr cell simulations (AREUS software) :
  - HL-LHC conditions with pile-up of 140.



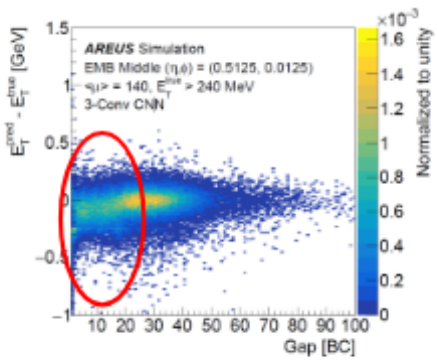
All methods outperform legacy algorithm.
Clear improvement of overlapping signals.



Legacy algorithm :
5 BC in the peak



3-Conv CNN:
5 BC in the peak; 8 in the past



Single cell LSTM :
5 BC in the peak; ∞ in the past

More receptive fields in time

FPGA Implementations

High speed evaluation => FPGA

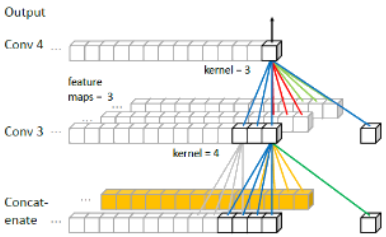
* set of weights (optimized in training),
* architecture (layers, dimensions, ...),
* mathematical operations.

* ALM : adaptive logic modules,
* DSP : digital signal processors,
* fixed-point arithmetic, LUT for non-linear functions.

$$\begin{pmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \end{pmatrix} = A\left(\begin{pmatrix} w_{11} & w_{12} & w_{13} \\ w_{21} & w_{22} & w_{23} \\ w_{31} & w_{32} & w_{33} \\ w_{41} & w_{42} & w_{43} \end{pmatrix} \times \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} + \begin{pmatrix} b_1 \\ b_2 \\ b_3 \\ b_4 \end{pmatrix}\right)$$

Activation function for non-linear element operations

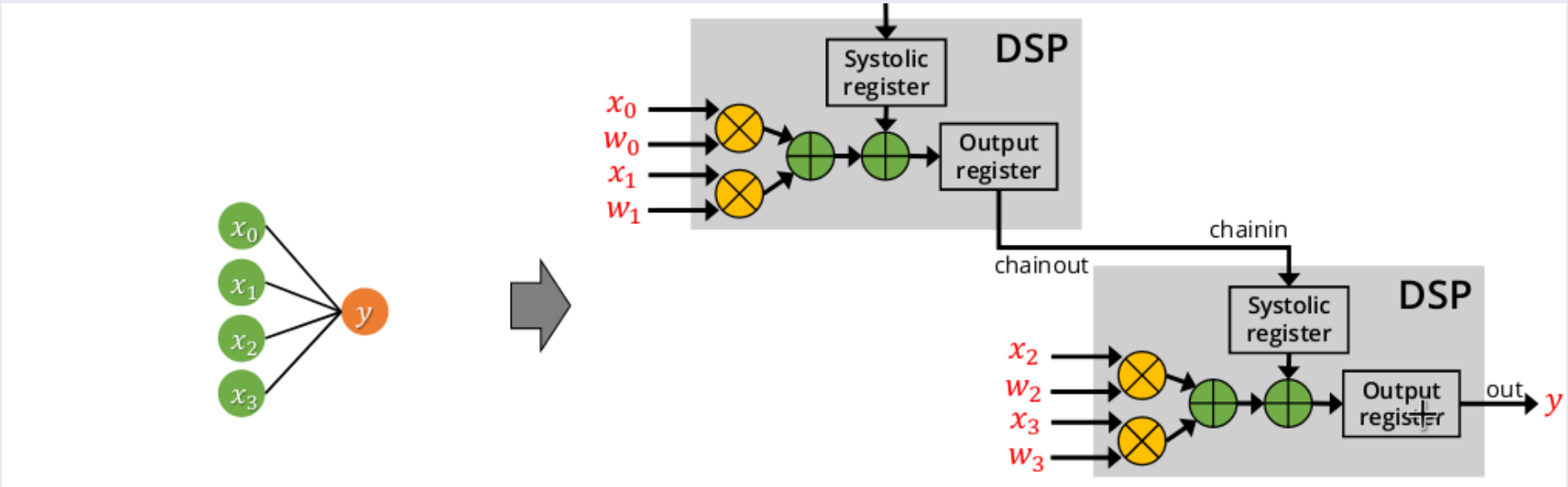
FPGA Implementations: CNNs



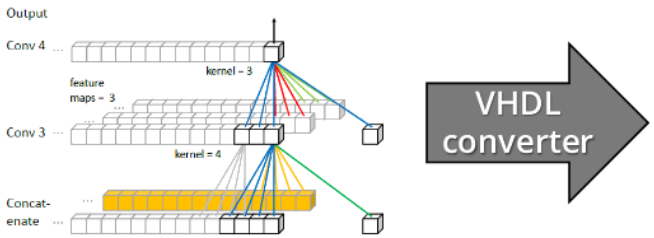
VHDL converter

- Configured by Keras models
- Optimised for low latency
  - CNN architecture mapped to DSPs chains
  - pipelined inputs

Maximize DSP usage (DSP chains) for fast weight multiplication

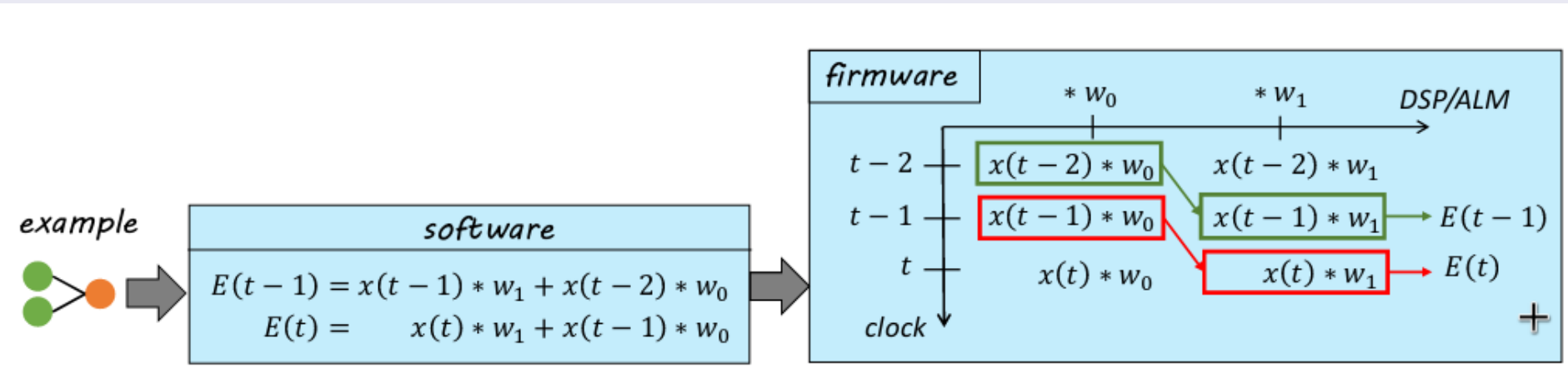


FPGA Implementations: CNNs



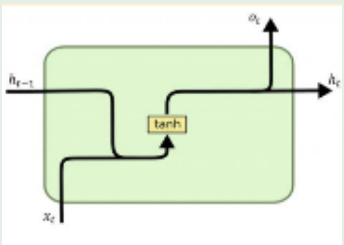
- Configured by Keras models
- Optimised for low latency
  - CNN architecture mapped to DSPs chains
  - pipelined inputs

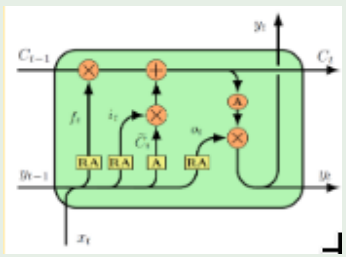
Input pipeline : reuse hardware as soon as available to deal with continuous flow of data.



FPGA Implementations: RNNs

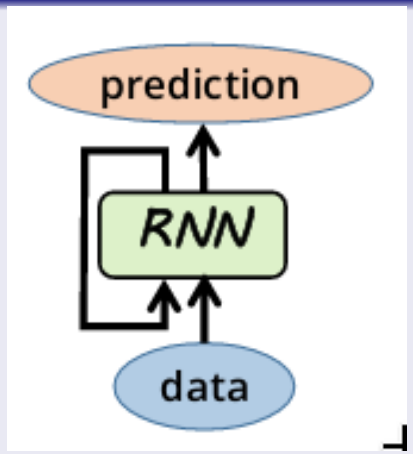
Template RNN functions
(configured by Keras model)





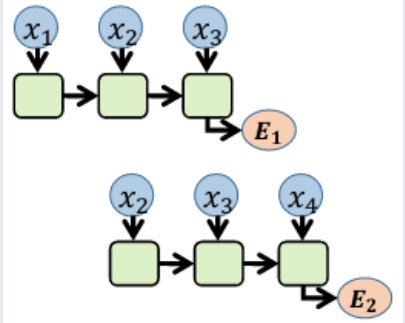
"Fully unrolled"
(parallelize all possible tasks,
allow compiler optimize for
high frequency)

Single Cell HLS



Single RNN instance on
hardware
Wait output of previous cell
(arriving at frequency/latency)

Sliding window HLS



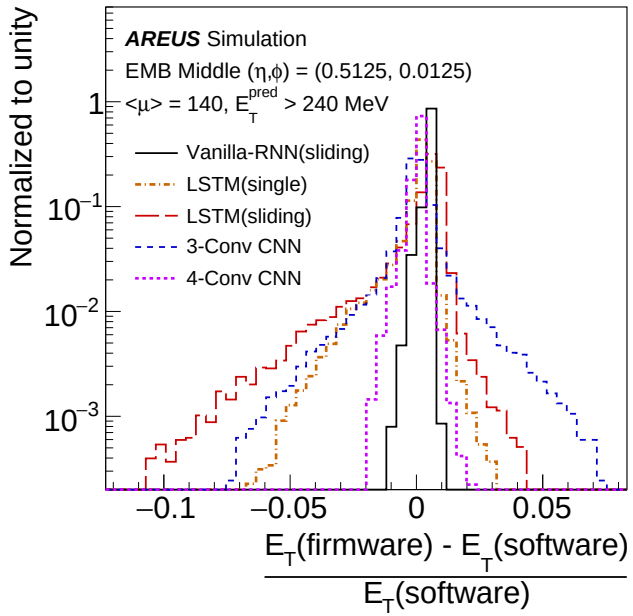
5 RNN instances (3 in sketch)
Independent sequences
Pipelined

Implemented in HLS to improve flexibility

FPGA Implementations: Results

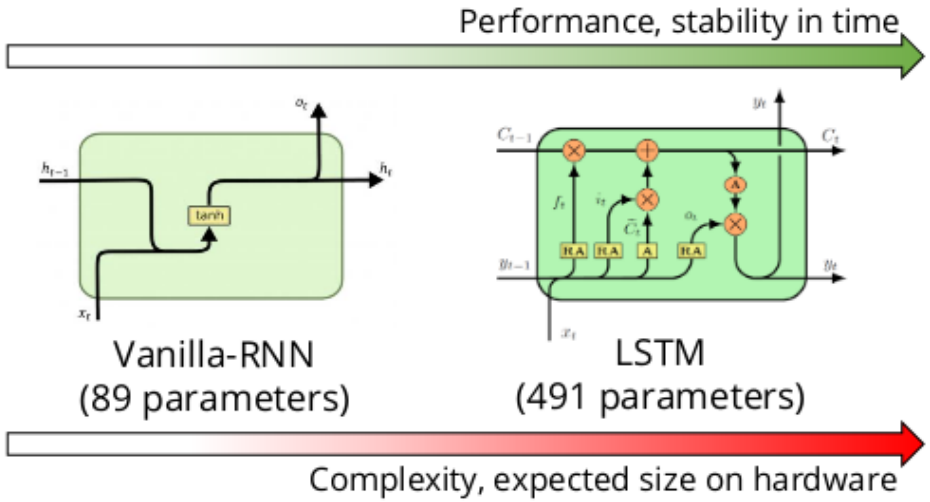
- Compare Intel Stratix10 simulation (Quartus 21.1 and Modelsim10.7c) to Keras/Tensorflow:
  - Pulse samples from AREUS LArcell data

- Optimized fixed-point and LUT representations :
  - minimize resources VS compatibility software/firmware
  - 18 bits total (Stratix10 : 18x19 DSP)
    - 10 decimal for CNNs
    - 13 decimal for RNNs



acceptable resolution from the quantification , lower than what is expected from the noise

DSP usage, theory



- LSTM Network
  - 4 Gates/cell (0 for first since recurrent state = 0)
  - 7 Vector to Vector or scalar to vector multiplication per cell
  - 1 Vector multiplication for output
  - TOTAL: $4 * Hdim^2 * (Ncell - 1) + 7 * Hdim * Ncell + Hdim$
- Vanilla
  - 1 Gates/cell (0 for first since recurrent state = 0)
  - 1 scalar to vector multiplication per cell
  - 1 Vector to Vectormultiplication for output
  - TOTAL: $Hdim^2 * (Ncell - 1) + Hdim * Ncell + Hdim$

Results

Number of channels that can fit in Stratix 10 SX FPGA (DSP/channel)

LSTM

Hidden Dimensions	Multiplexing 1 40 MHz	2 80 MHz	3 120 MHz	4 160 MHz	5 200 MHz	6 240 MHz	7 280 MHz	8 320 MHz	9 360 MHz	10 400 MHz	11 440 MHz	12 480 MHz	13 520 MHz	14 560 MHz	15 600 MHz
2	84 (136.0)	169 (68.0)	254 (45.3)	338 (34.0)	423 (27.2)	508 (22.7)	592 (19.4)	677 (17.0)	762 (15.1)	847 (13.6)	931 (12.4)	1016 (11.3)	1101 (10.5)	1185 (9.7)	1270 (9.1)
3	45 (252.0)	91 (126.0)	137 (84.0)	182 (63.0)	228 (50.4)	274 (42.0)	320 (36.0)	365 (31.5)	411 (28.0)	457 (25.2)	502 (22.9)	548 (21.0)	594 (19.4)	640 (18.0)	685 (16.8)
4	28 (400.0)	57 (200.0)	86 (133.3)	115 (100.0)	144 (80.0)	172 (66.7)	201 (57.1)	230 (50.0)	259 (44.4)	288 (40.0)	316 (36.4)	345 (33.3)	374 (30.8)	403 (28.6)	431 (26.7)
5	19 (580.0)	39 (290.0)	59 (193.3)	79 (145.0)	99 (116.0)	119 (96.7)	139 (82.9)	158 (72.5)	178 (64.4)	198 (58.0)	218 (52.7)	238 (48.3)	258 (44.6)	278 (41.4)	297 (38.7)
6	14 (792.0)	29 (396.0)	43 (264.0)	58 (198.0)	72 (158.4)	87 (132.0)	101 (113.1)	116 (99.0)	130 (88.0)	145 (79.2)	160 (72.0)	174 (66.0)	189 (60.9)	203 (56.6)	218 (52.8)
7	11 (1036.0)	22 (518.0)	33 (345.3)	44 (259.0)	55 (207.2)	66 (172.7)	77 (148.0)	88 (129.5)	100 (115.1)	111 (103.6)	122 (94.2)	133 (86.3)	144 (79.7)	155 (74.0)	166 (69.1)
8	8 (1312.0)	17 (656.0)	26 (437.3)	35 (328.0)	43 (262.4)	52 (218.7)	61 (187.4)	70 (164.0)	79 (145.8)	87 (131.2)	96 (119.3)	105 (109.3)	114 (100.9)	122 (93.7)	131 (87.5)
9	7 (1620.0)	14 (810.0)	21 (540.0)	28 (405.0)	35 (324.0)	42 (270.0)	49 (231.4)	56 (202.5)	64 (180.0)	71 (162.0)	78 (147.3)	85 (135.0)	92 (124.6)	99 (115.7)	106 (108.0)
10	5 (1960.0)	11 (980.0)	17 (653.3)	23 (490.0)	29 (392.0)	35 (326.7)	41 (280.0)	47 (245.0)	52 (217.8)	58 (196.0)	64 (178.2)	70 (163.3)	76 (150.8)	82 (140.0)	88 (130.7)
11	4 (2332.0)	9 (1166.0)	14 (777.3)	19 (583.0)	24 (466.4)	29 (388.7)	34 (333.1)	39 (291.5)	44 (259.1)	49 (233.2)	54 (212.0)	59 (194.3)	64 (179.4)	69 (166.6)	74 (155.5)
12	4 (2736.0)	8 (1368.0)	12 (912.0)	16 (684.0)	21 (547.2)	25 (456.0)	29 (390.9)	33 (342.0)	37 (304.0)	42 (273.6)	46 (248.7)	50 (228.0)	54 (210.5)	58 (195.4)	63 (182.4)
13	3 (3172.0)	7 (1586.0)	10 (1057.3)	14 (793.0)	18 (634.4)	21 (528.7)	25 (453.1)	29 (396.5)	32 (352.4)	36 (317.2)	39 (288.4)	43 (264.3)	47 (244.0)	50 (226.6)	54 (211.5)
14	3 (3640.0)	6 (1820.0)	9 (1213.3)	12 (910.0)	15 (728.0)	18 (606.7)	22 (520.0)	25 (455.0)	28 (404.4)	31 (364.0)	34 (330.9)	37 (303.3)	41 (280.0)	44 (260.0)	47 (242.7)

Vanilla

Hidden Dimensions	Multiplexing 1 40 MHz	2 80 MHz	3 120 MHz	4 160 MHz	5 200 MHz	6 240 MHz	7 280 MHz	8 320 MHz	9 360 MHz	10 400 MHz	11 440 MHz	12 480 MHz	13 520 MHz	14 560 MHz	15 600 MHz
2	411 (28.0)	822 (14.0)	1234 (9.3)	1645 (7.0)	2057 (5.6)	2468 (4.7)	2880 (4.0)	3291 (3.5)	3702 (3.1)	4114 (2.8)	4525 (2.5)	4937 (2.3)	5348 (2.2)	5760 (2.0)	6171 (1.9)
3	213 (54.0)	426 (27.0)	640 (18.0)	853 (13.5)	1066 (10.8)	1280 (9.0)	1493 (7.7)	1706 (6.8)	1920 (6.0)	2133 (5.4)	2346 (4.9)	2560 (4.5)	2773 (4.2)	2986 (3.9)	3200 (3.6)
4	130 (88.0)	261 (44.0)	392 (29.3)	523 (22.0)	654 (17.6)	785 (14.7)	916 (12.6)	1047 (11.0)	1178 (9.8)	1309 (8.8)	1440 (8.0)	1570 (7.3)	1701 (6.8)	1832 (6.3)	1963 (5.9)
5	88 (130.0)	177 (65.0)	265 (43.3)	354 (32.5)	443 (26.0)	531 (21.7)	620 (18.6)	708 (16.2)	797 (14.4)	886 (13.0)	974 (11.8)	1063 (10.8)	1152 (10.0)	1240 (9.3)	1329 (8.7)
6	64 (180.0)	128 (90.0)	192 (60.0)	256 (45.0)	320 (36.0)	384 (30.0)	448 (25.7)	512 (22.5)	576 (20.0)	640 (18.0)	704 (16.4)	768 (15.0)	832 (13.8)	896 (12.9)	960 (12.0)
7	48 (238.0)	96 (119.0)	145 (79.3)	193 (59.5)	242 (47.6)	290 (39.7)	338 (34.0)	387 (29.8)	435 (26.4)	484 (23.8)	532 (21.6)	580 (19.8)	629 (18.3)	677 (17.0)	726 (15.9)
8	37 (304.0)	75 (152.0)	113 (101.3)	151 (76.0)	189 (60.8)	227 (50.7)	265 (43.4)	303 (38.0)	341 (33.8)	378 (30.4)	416 (27.6)	454 (25.3)	492 (23.4)	530 (21.7)	568 (20.3)
9	30 (378.0)	60 (189.0)	91 (126.0)	121 (94.5)	152 (75.6)	182 (63.0)	213 (54.0)	243 (47.2)	274 (42.0)	304 (37.8)	335 (34.4)	365 (31.5)	396 (29.1)	426 (27.0)	457 (25.2)
10	25 (460.0)	50 (230.0)	75 (153.3)	100 (115.0)	125 (92.0)	150 (76.7)	175 (65.7)	200 (57.5)	225 (51.1)	250 (46.0)	275 (41.8)	300 (38.3)	325 (35.4)	350 (32.9)	375 (30.7)
11	20 (550.0)	41 (275.0)	62 (183.3)	83 (137.5)	104 (110.0)	125 (91.7)	146 (78.6)	167 (68.8)	188 (61.1)	209 (55.0)	230 (50.0)	251 (45.8)	272 (42.3)	293 (39.3)	314 (36.7)
12	17 (648.0)	35 (324.0)	53 (216.0)	71 (162.0)	88 (129.6)	106 (108.0)	124 (92.6)	142 (81.0)	160 (72.0)	177 (64.8)	195 (58.9)	213 (54.0)	231 (49.8)	248 (46.3)	266 (43.2)
13	15 (754.0)	30 (377.0)	45 (251.3)	61 (188.5)	76 (150.8)	91 (125.7)	106 (107.7)	122 (94.2)	137 (83.8)	152 (75.4)	168 (68.5)	183 (62.8)	198 (58.0)	213 (53.9)	229 (50.3)
14	13 (868.0)	26 (434.0)	39 (289.3)	53 (217.0)	66 (173.6)	79 (144.7)	92 (124.0)	106 (108.5)	119 (96.4)	132 (86.8)	145 (78.9)	159 (72.3)	172 (66.8)	185 (62.0)	199 (57.9)

Fit for all scenarios Fit for 2 or 3 FEB per LASP

Fit for 2 FEB per LASP Don't fit

FPGA Implementations: Ressources usage

For design with single channel input data:

	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
Resource Usage (per channel)					
DSP	46 0.8%	42 0.7%	34 0.6%	176 3.1%	738 12.8%
ALM	5684 0.6%	5702 0.6%	13115 1.4%	18079 1.9%	69892 7.5%

- 384 or 512 channels depending on the config choosen

For design with multiplexing input data:

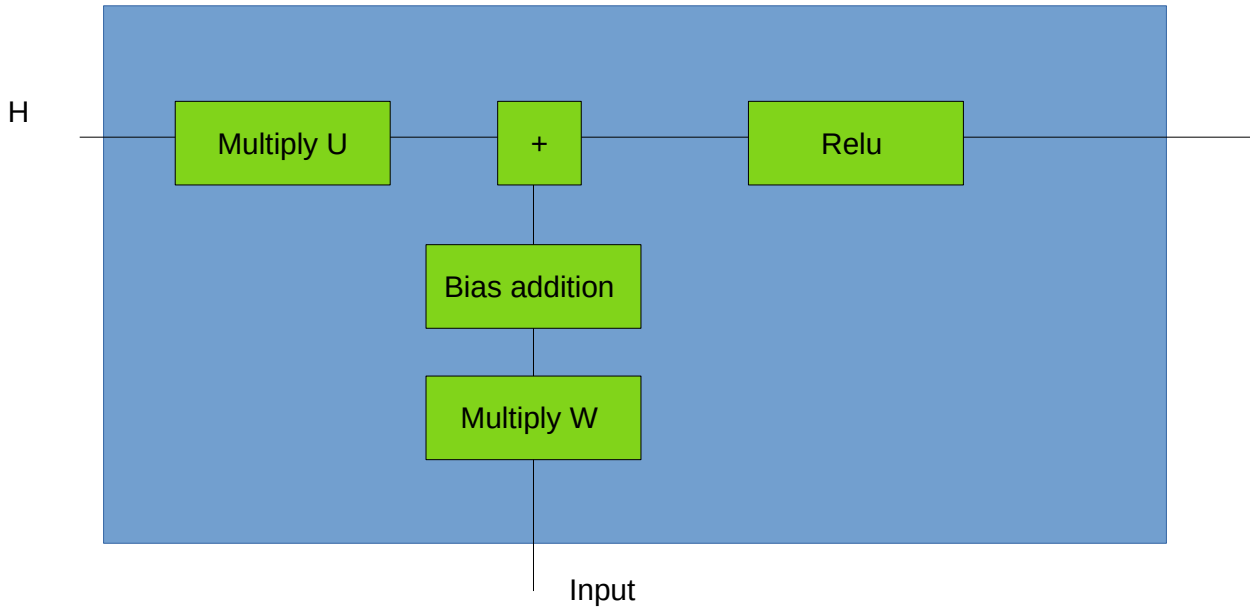
	3-Conv CNN	4-Conv CNN	Vanilla RNN
Multiplicity	6	6	15
Frequency F_{max} [MHz]	344	334	640
Latency clk_{core} cycles	81	62	120
Max. Channels	390	352	576
Resource Usage			
DSP	46 0.8%	42 0.7%	152 2.6%
ALMs	14235 1.5%	15627 1.7%	5782 0.6%

Challenging but multiplexing LAr channels could allow CNN or Vanilla-RNN usage

Conclusion

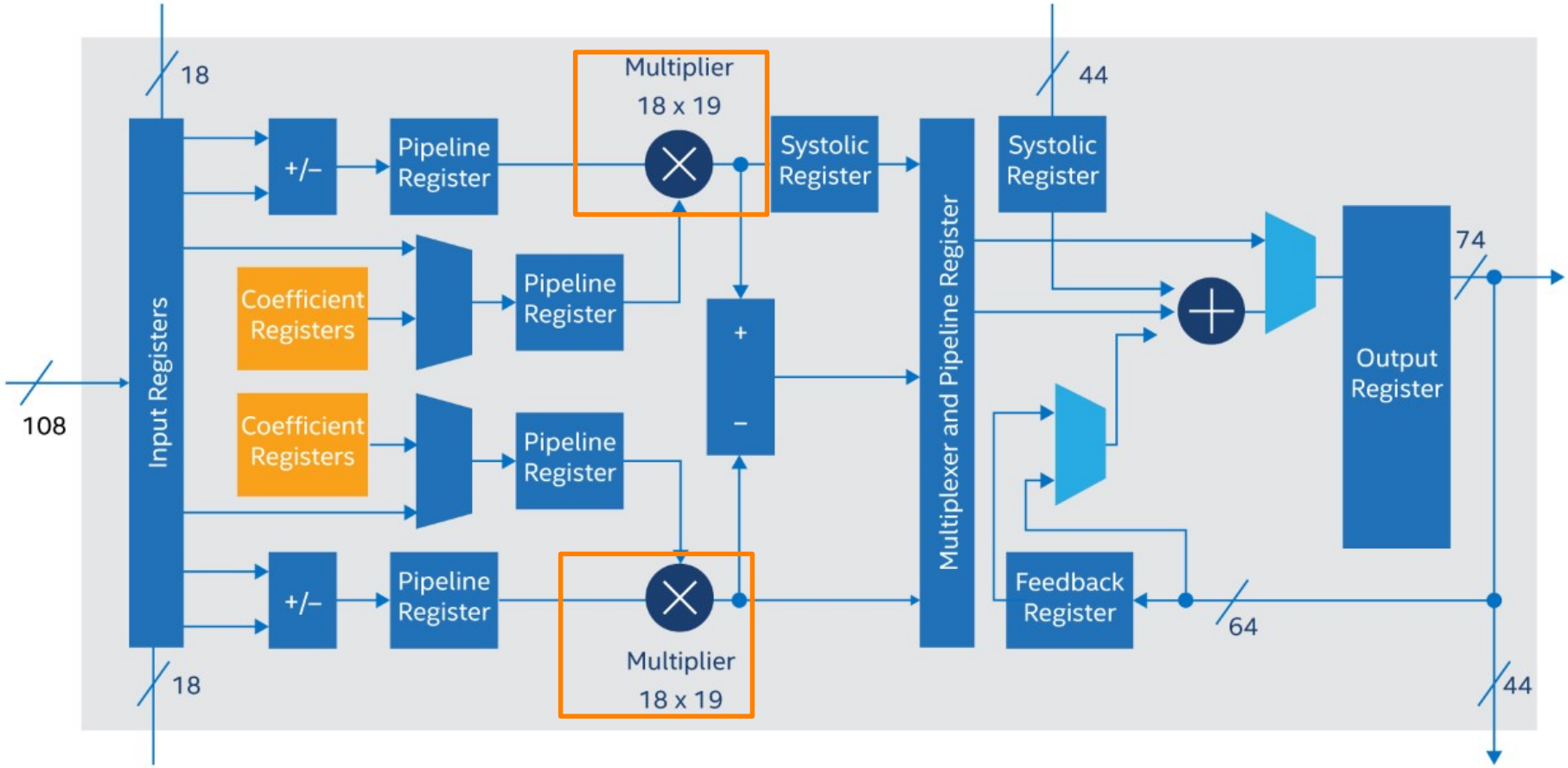
- HL-HC will require improving ATLAS LAr energy measurements
  - 2 new methods using neural networks: CNN based and RNN based
- For both CNN/RNN several algorithms are developed
  - Focused on recovering energy resolution in high pileup environments by using information from past events
  - All methods outperform legacy algorithms in HL-LHC conditions
- FPGA implementation for fast processing:
  - CNN : dedicated VHDL
  - RNN : flexible HLS
  - Good reproduction of Keras results with firmware simulation
  - Optimizations ongoing to reduce resource usage and latency to fit ATLAS requirements
- First results provide a prototype firmware with an NN capable of improving the LAr energy reconstruction and fits into the FPGAs that will be used for the phase 2 data processing
- Aad, G., Berthold, AS., Calvet, T. et al. Artificial Neural Networks on FPGAs for Real-Time Energy Reconstruction of the ATLAS LAr Calorimeters. Comput Softw Big Sci 5, 19 (2021). https://doi.org/10.1007/s41781-021-00066-y

Optimisation exemple: vanCell



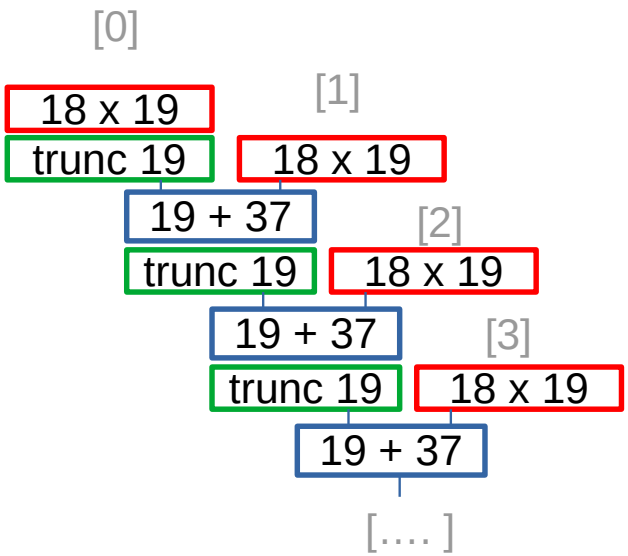
- Multiply U: Vector(H) and matrix(Weights) multiplication : n^2 Multiplication
- Multiply W: Vector(Weights) multiplied by scalar(Input) : n Multiplication
- Multiply U is the biggest function of the Vanilla: optimisation focused on it

Optimisation exemple: Vector multiplication



Intel® Stratix® 10 Device DSP Block: Standard-Precision Fixed Point

Optimisation exemple: Vector multiplication




```
fp18 weights[8];
fp19 data[8];
fp19 acc;
for (int i=0; i<8 , i++){
  acc += weights[i] * data[i];
}
```

18b x 19b = 37b multiplier

37b + 19b = 37b adder truncated 19b

Latency = latency DSP + (latency adder x 7)
DSP : 8 mul = 4 DSP
ALUT : 7x add[19+37] =7*37 = 259
a

Optimisation exemple: Vector multiplication

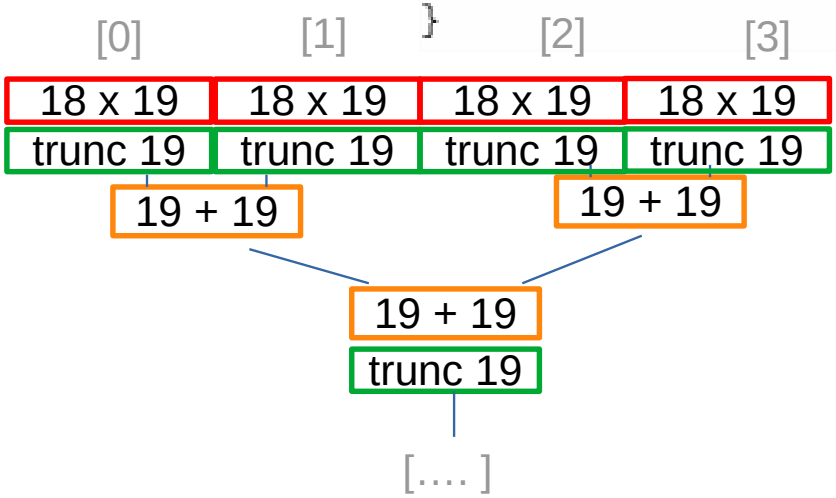
```
fp18 weights[8];
fp19 data[8];
fp19 acc;
for (int i=0; i<8 , i++){

  acc += (fp19) weights[i] * data[i];

}
```

18b x 19b = 37b multiplier truncated 19b

19b + 19b = 19b adder



Latency = latency DSP + (latency adder x 3)
-4 since slide before
DSP : 8 mul = 4 DSP
ALUT : 7x add[19+19] =7*19=133
-126 since slide before

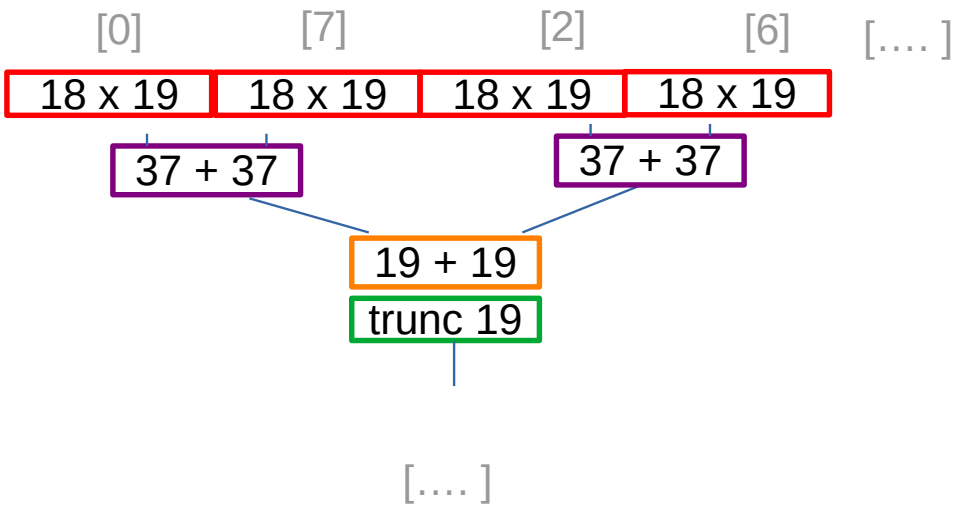
Optimisation exemple: Vector multiplication

```
fp18 weights[8];
fp19 data[8];
fp19 acc;
for (int i=0; i< 4 , i++){
  acc += (fp19) ( weights[i] * data[i] + weights[7-i] * data[7-i] ) ;
}
```

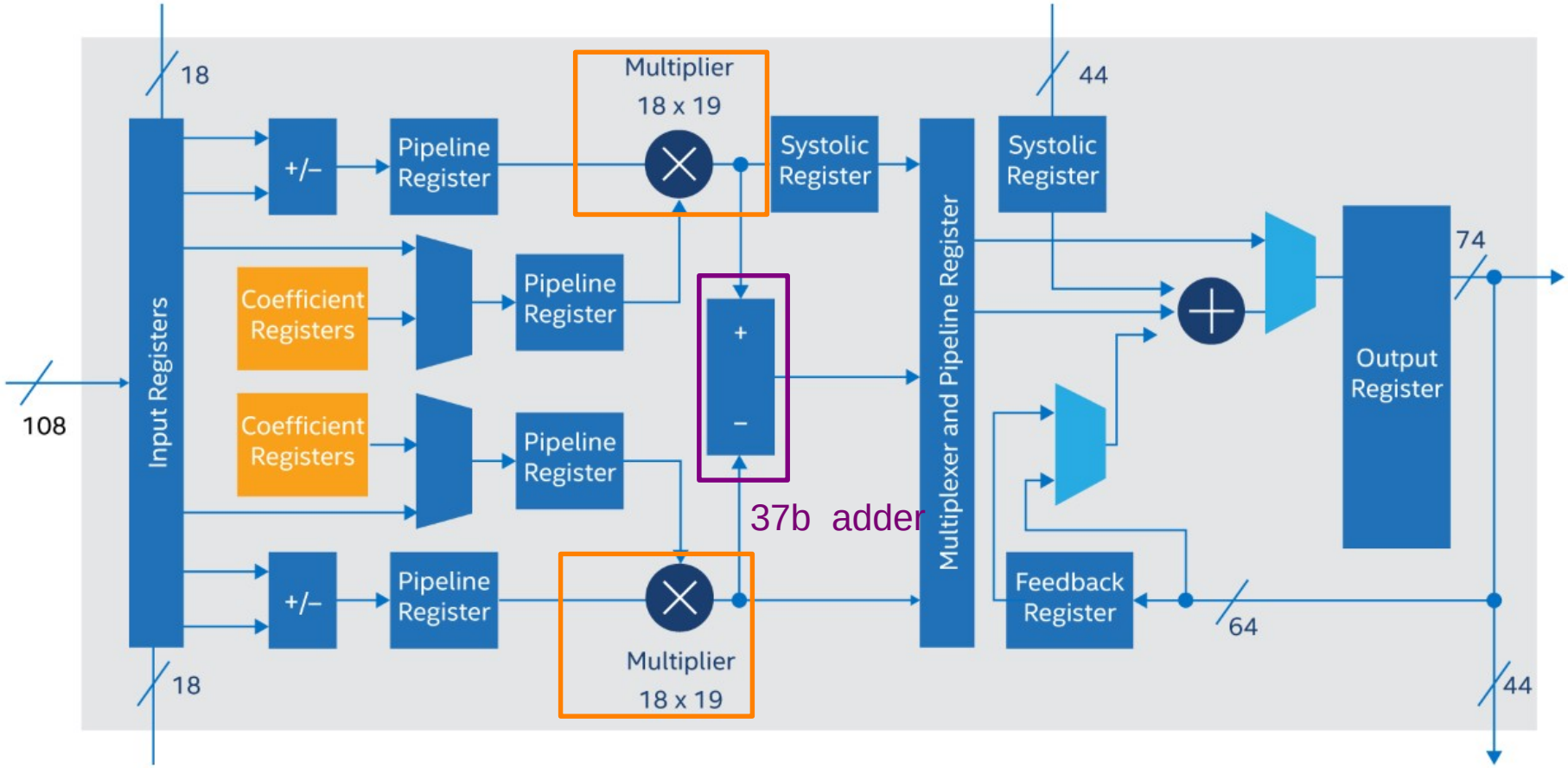
18b x 19b = 37b multipliers

19b + 19b = 19b adder

37b + 37b = 37b adder



Optimisation exemple: Vector multiplication



Intel® Stratix® 10 Device DSP Block: Standard-Precision Fixed Point

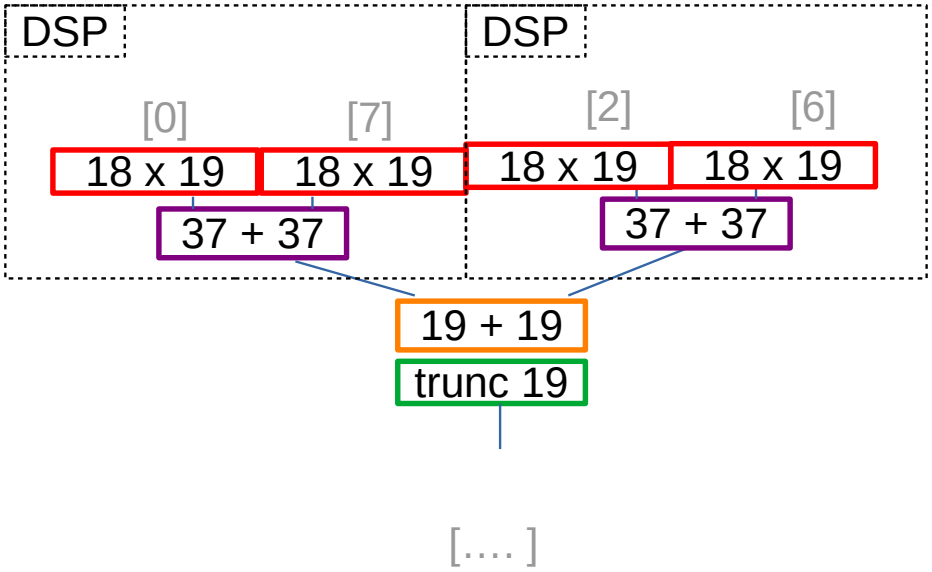
Optimisation exemple: Vector multiplication

```
fp18 weights[8];
fp19 data[8];
fp19 acc;
for (int i=0; i< 4 , i++){
  acc += (fp19) ( weights[i] * data[i] + weights[7-i] * data[7-i] ) ;
}
```

18b x 19b = 37b multipliers

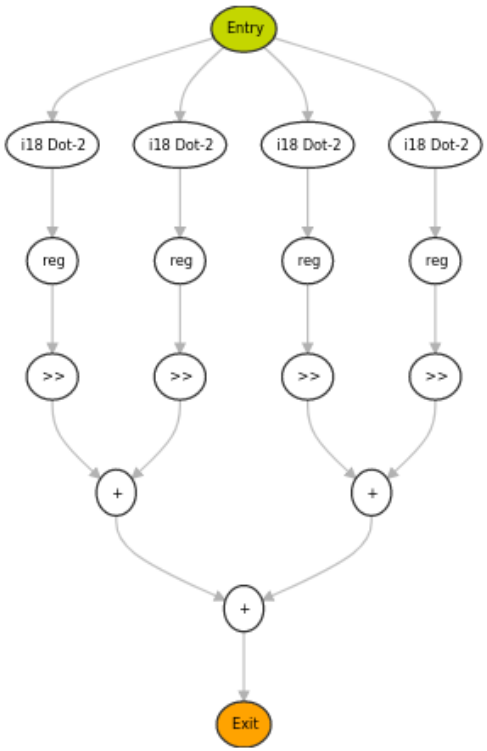
19b + 19b = 19b adder

37b + 37b = 37b adder



Latency = latency DSP + (latency adder x 2)
-1 since slide before
DSP : 8 mul + acc = 4 DSP
ALUT : 3x add[19+19] =3*19=57
-76 since slide before

Optimisation conclusion



	ALUT	FFs	RAM	MLAB	DSP
▼ main.cpp:175 > main.cpp:125 > main.cpp:112	25	49	0	0	1.33333
18bit Integer Dot Product of Size 2 (x4)	25	49	0	0	1.33333
▼ main.cpp:175 > main.cpp:125 > main.cpp:112	51	99	0	0	2.66667
18bit Integer Dot Product of Size 2 (x8)	51	99	0	0	2.66667
▼ main.cpp:175 > main.cpp:133	57	0	0	0	0
19-bit Integer Add (x3)	57	0	0	0	0

- DSP expected
- ALUTS expected for addition
- ALUTs: 76 In the multiplication $= 8 * 9.5 = 8 * 19/2$: Register in ALUT before DSP ?

Optimisation conclusion

- Optimisation: -70% of ALUTs for same performance
- HLS is cpp but it's needed to understand how the fpga work
- Functions have to be optimised as small block separate by register (directive in hls)
- Depend on the FPGA
- Intel HLS compiler
- Integration in hls4ml