A Convolutional Neural Network (CNN) for handwritten digit recognition on FPGA using HLS

Giacomo Boldini
giacomo.boldini@studenti.unipr.it

University of Parma
Master's Degree in Computer Science
Embedded Systems

AA 2021/22

Project repository here
[1]

Goals & Outline

Goals

- Creation of a NN for handwritten digit classification.
- Implementation of the NN on FPGA using HLS/Vivado.
- Prove that HW solutions is faster than SW (C) solutions.

Outline:

1. Python: Create and train NN model
2. C: NN implementation
3. Vitis/C++: NN synthesis and validation
4. Conclusions

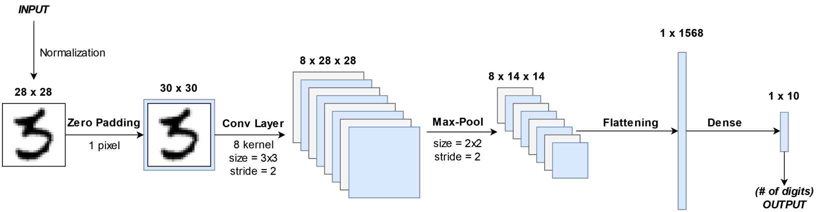
The (C)NN model

CNN architecture choosed:

- image-processing task;
- no need of manual feature extraction: done automatically;
- less number of parameters than other NNs.

API: Python Keras/Tensorflow [2].

Model as simple as possible:



Training

TrainX shape = (60000, 28, 28)
Training epochs = 10 (empiric)

Layers' trainable parameters:

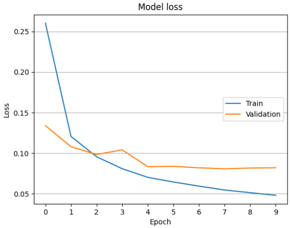
Layer (type)	Output Shape	Param #
ZeroPadding2D	(30, 30, 1)	0
Conv2D	(28, 28, 8)	80
MaxPooling2D	(14, 14, 8)	0
Flatten	(1568)	0
Dense	(10)	15690
TOT		**15770**

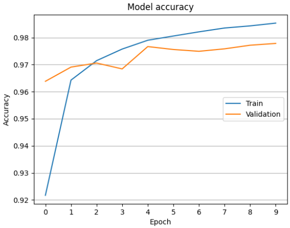
Accuracy:

- validation set (20% of test set): 97.78%
- test set (#10000 samples): **98.070%**

Mean time for a prediction: ∼**35 ms**

Training history:

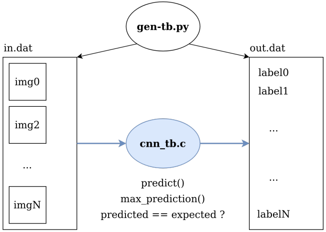




C cnn() function

```
void cnn(float img_in [IMG_ROWS][IMG_COLS], float prediction[DIGITS])
{
  // Normalization and padding.
  float pad_img [PAD_IMG_ROWS][PAD_IMG_COLS] = { 0 };
  normalization_and_padding(img_in, pad_img);
  // Convolution.
  float features [FILTERS][IMG_ROWS][IMG_COLS] = { 0 };
  convolutional_layer(pad_img, features);
  // Pooling.
  float pool_features [FILTERS][POOL_IMG_ROWS][POOL_IMG_COLS] = { 0 };
  max_pooling_layer(features, pool_features);
  // Flattening.
  float flat_array [FLAT_SIZE] = { 0 };
  flattening_layer(pool_features, flat_array);
  // Dense.
  dense_layer(flat_array, prediction);
}
```

C `main()` / testbench



- MNIST TestX samples: 10000 N: 100 ∼ 250
- Accuracy: $\frac{\text{correct predictions}}{\text{total predictions}}$ Test successfull ⇔ Accuracy ≥ 95%

Mean time for a prediction:

- **0.82 ms** - O0 (∼40x faster than Python)
- **0.17 ms** - O3 (∼200x faster than Python)

Code optimizations for Vitis/FPGA

C implementation not optimized for Vitis/FPGA deployment.

CNN parallelism

1. CNN creates implicit parallelism on filters.
2. CNN does not need all the data from the previous layer to start computing the output response for the current layer.

Optimize code:

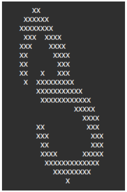
- `hls::stream` [3] between functions: FIFO with blocking API `read()` and `write()`.
  1. + new function `dataflow_section(img1,img2,...,img8)` that clones input image `FILTER_NUMBER` times.
  2. + sw chages: eg. convolution with sliding-window.

C simulation

Total predictions: 500.
Correct predictions: 98.20 % → **OK**.
Average latency: 2.33 ms → a little bit more than C.

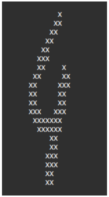
Some bad classifications:

(images normalized and rounded)



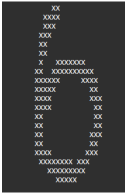
Expected: **3**

Got:
0: 0.000002
1: 0.000000
2: 0.001373
3: 0.213332
4: 0.000003
5: 0.000935
6: 0.000000
7: 0.000000
8: 0.783027
9: 0.001329



Expected: **4**

Got:
0: 0.000000
1: 0.000045
2: 0.000020
3: 0.000661
4: 0.253086
5: 0.000059
6: 0.000414
7: 0.000036
8: 0.000321
9: 0.745357



Expected: **6**

Got:
0: 0.735325
1: 0.000000
2: 0.000000
3: 0.000000
4: 0.000000
5: 0.000019
6: 0.264633
7: 0.000000
8: 0.000004
9: 0.000020

C synthesis I

Common parameters:

- Target device: **xc7a200tfbg484-1**
- Target clock period: **10ns** (clock freq.: 100 *MHz*)

Different *"levels of optimization"* (directives):

1. No directives

Target	Estimated	Uncertainty
10.00 ns	7.300 ns	2.70 ns

Modules & Loops	Issue Type	Violation Type	Distance	Slack	Latency(cycles)	Latency(ns)	Iteration Latency	Interval	Trip Count	Pipelined	BRAM	DSP	FF	LUT	URAM
cnn				-	361721	3,617E6	-	361722	-	no	231	69	16679	18273	0
dataflow_section				-	342698	3,427E6	-	342698	-	no	215	69	16409	17588	0
pad_for_rows				-	17160	1,720E5	572	-	30	no	-	-	-	-	-
VITIS_LOOP_102_1				-	1860	1,860E4	62	-	30	no	-	-	-	-	-

C synthesis II

❷ Default directives

Target	Estimated	Uncertainty
10.00 ns	7.300 ns	2.70 ns

Modules & Loops	Issue Type	Violation Type	Distance	Slack	Latency(cycles)	Latency(ns)	Iteration Latency	Interval	Trip Count	Pipelined	BRAM	DSP	FF	LUT	URAM
cnn				-	19523	1,950E5	-	19524	-	no	264	104	40089	30930	0
cnn_Pipeline_pad_for_rows_pad_for_cols				-	920	9,200E3	-	920	-	no	0	0	1018	419	0
cnn_Pipeline_VITIS_LOOP_102_1_VITIS_LOOP_103_2				-	903	9,030E3	-	903	-	no	0	0	45	167	0
dataflow_section	II Violation			-	17695	1,770E5	-	17695	-	no	248	104	39017	29916	0

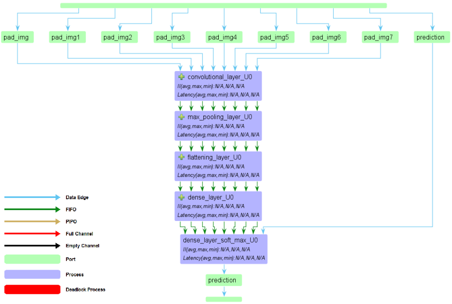
❸ Dataflow directive

Target	Estimated	Uncertainty
10.00 ns	7.241 ns	2.70 ns

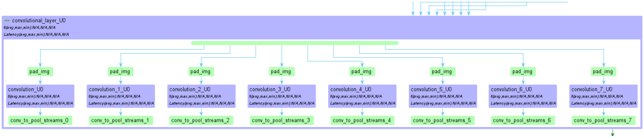
Modules & Loops	Issue Type	Violation Type	Distance	Slack	Latency(cycles)	Latency(ns)	Iteration Latency	Interval	Trip Count	Pipelined	BRAM	DSP	FF	LUT	URAM
cnn				-	5825	5,825E4	-	5826	-	no	384	129	46902	38213	0
cnn_Pipeline_pad_for_rows_pad_for_cols				-	920	9,200E3	-	920	-	no	0	0	1018	419	0
cnn_Pipeline_VITIS_LOOP_102_1_VITIS_LOOP_103_2				-	903	9,030E3	-	903	-	no	0	0	45	167	0
dataflow_section				-	3997	3,997E4	-	3998	-	dataflow	368	129	45828	37234	0
convolutional_layer				-	3997	3,997E4	-	3998	-	dataflow	0	80	20585	18273	0
max_pooling_layer				-	792	7,920E3	-	793	-	dataflow	0	0	3513	4049	0
flattening_layer				-	198	1,980E3	-	199	-	dataflow	0	0	353	825	0
dense_layer				-	2013	2,013E4	-	2014	-	dataflow	320	40	15584	9632	0
dense_layer_soft_max				-	871	8,710E3	-	871	-	no	0	9	1071	1689	0

C synthesis III

Dataflow view:



(zoom on convolutional_layer)



Validation and implementation

C/RTL Cosimulation → **OK**

Modules & Loops	Avg II	Max II	Min II	Avg Latency	Max Latency	Min Latency
cnn	6747	6747	6747	6746	6746	6746
cnn_Pipeline_pad_for_rows_pad_for_cols	6747	6747	6747	918	918	918
cnn_Pipeline_clone_for_rows_clone_for_cols	6747	6747	6747	901	901	901
dataflow_section	6747	6747	6747	4922	4922	4922

```
Total predictions:  100
Correct predictions:  99.00 %
Average latency:  0.290000 (ms)
*** C/RTL co-simulation finished:  PASS ***
```

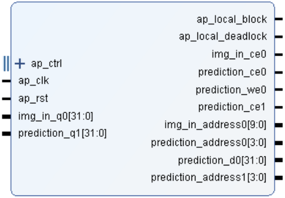
→ prediction time: **0.067** ms

Implementation (Vivado)

	Verilog
SLICE	12940
LUT	26381
FF	38178
DSP	129
BRAM	224
URAM	0
LATCH	0
SRL	1007
CLB	0

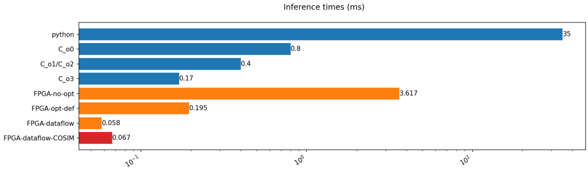
	Verilog
CP required	10.000
CP achieved post-synthesis	8.123
CP achieved post-implementation	9.449

Timing met



Conclusions

Main goal reached: **HW faster than SW - but not always**.



As future works:

- small SW changes could improve parallelism;
- more targeted Vitis pragmas could improve performance;
- using *fixed-point* arithmetic could reduce area **(* performance too)**;
- `grid-search` on NN architecture could increase accuracy (more performance) and reduce FPGA area (less price).

Thank you for your attention.

References

[1] *Github: HLS-CNN*. [Project repository]. URL: https://github.com/FedericoSerafini/HLS-CNN.

[2] Francois Chollet et al. *Keras*. 2015. URL: https://github.com/fchollet/keras.

[3] *Vitis High-Level Synthesis User Guide: HLS Stream Library*. [Online; visited june-2022]. URL: https://docs.xilinx.com/r/en-US/ug1399-vitis-hls/HLS-Stream-Library.

[4] Duda S. *How to Implement a Convolutional Neural Network Using High Level Synthesis*. Ed. by amiq.com. [Online; posted 14-December-2018]. 2018. URL: https://www.amiq.com/consulting/2018/12/14/how-to-implement-a-convolutional-neural-network-using-high-level-synthesis/.