https://gitlab.polytech.unice.fr/nn012527/hls_ia_team11

Embedded AI on FPGA using High Level Synthesis

GSE5 Embedded Systems Project 2020-2021

Prepared by team 11

Chaitanya GORE
Bogdan-Mihai NISTOR
Nelli NYISZTOR

Table of Contents

Introduction

The goal of the project was to understand and implement a hardware accelerated version of a specific type of CNN (LeNet-5) using high-level synthesis and present a quantitative analysis of performance and energy efficiency improvements. The target FPGA device is a ZedBoard Zynq-7000 SoC. First, we implemented the main neural network layers in C, optimized and modified our code in order to make it synthesizable with Vivado HLS. As a second step, we introduced HLS pragmas and added parallelization so that we could increase resource usage (balance between e.g. BRAM and LUT resources).

After that we used SDSoC to compile our code, obtain the image, the bootloader and the executable file for several implementations, to be able to test the performance on the target device. In order to analyze the power and energy consumption, after generating the RTL design with Vivado HLS, we ran the implementation with Vivado. Our entire solution can be found on GitLab, along with a demo video, detailed synthesis results, as for for file structure, please refer to README.md.

LeNet-5 layers

In order to do our neural network, we have followed the LeNet 5 architecture for handwritten and machine printed character recognition. The purpose is to reduce the images to a form which is easier to process, without losing features which are critical to getting a good prediction. The architecture of LeNet-5 is quite simple, as it implies 2 convolutional layers, 2 pooling layers, two fully connected layers and finally a softmax classifier. The relationship between these functions can be seen represented in the following section.

Convolution layer: The purpose of this first function is to extract the high-level features (e.g. edges, gradient orientation) from the input images. For the first convolution layer we have used the Kernel in order to shift over our image pixels, performing the multiplication between *kernel* and the portion *imgPart* of the matrix, the results being added up in a sole result. This process is done through the *sumProduct* function. The kernel moves left to right, top to bottom until the entire matrix is traversed.

In the second convolutional layer however, we also used the Rectified Linear Unit (ReLU) in order to improve and increase the non-linear features in the images we have.

Pooling layer: We have used it in order to reduce the size of the convolved feature so that we could lower the computational power required to process the data while at the same time extract the dominant features. For this, we have used the Max Pooling method which has a double benefit: noise reduction and dimensionality reduction.

Fully Connected Layer: The role is of classification of the images into labels after receiving the results from the pooling and convolution layers. In the first fully connected layer function, the input from the second Pooling layer is being flattened, therefore the values being put into a vector. Furthermore, for each fully connected layer we have multiplied the *input* with the *kernel* and added the results together. In the end we have added the bias to the final result and used once again the ReLU function.

For the second fully connected layer function however, we didn't use ReLU anymore, but after passing through this final layer, we used the **SoftMax** activation function. This is a specific function that is used in order to take the probabilities of its input that are in a particular classification. In the end, after the passing through SoftMax, we have all the probabilities needed of the images so that we can make an accurate assumption.

Functional overview, high-level design

In this chapter we can see the visual representation and relationship of the LeNet-5 functions presented above. We can also see how the matrix dimension changes from function to function after processing.

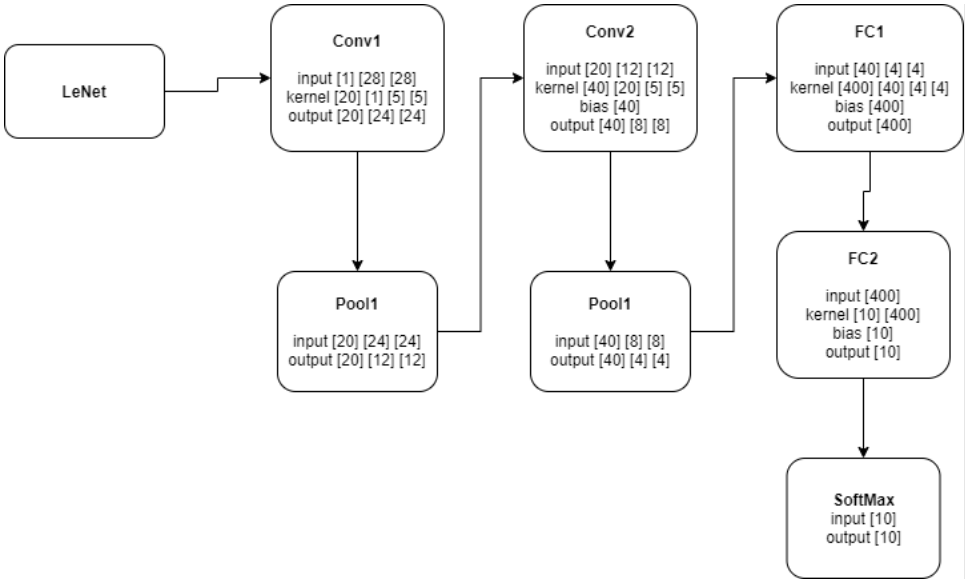


Figure 1. LeNet-5 high-level design

Hardware

The hardware plays key role for the performance of the system. As mentioned above, our target device is ZedBoard platform featuring a Zynq-7000 SoC, which is tightly coupled with ARM dual core Cortex A9/ Xilinx FPGA. The processor runs on 667MHz, whereas the FPGA operates with 100MHz, this results in a **6.7 factor increase in frequency**. This could be an interesting additional aspect when we consider the final results from energy efficiency point of view.

An array in C code is implemented by memory in RTL. The array can be targeted to any memory resource on the device. Arrays can be merged with other arrays and reconfigured. Arrays can be partitioned into individual elements (Implemented by smaller RAMs or registers).

Software, Test and results

Implementation stages

FLOAT: Initially, we received a floating-point implementation of the above mentioned LeNet5 CNN. Since the crucial .c files were not at our disposal, we needed to write our own logic for the convolution, pooling and fully connected layers. (Run time on our own PC: 14s+)

FIXED: HLS compliant code operates with fixed point arithmetic to avoid complex operations requiring FPU. Moreover, we aimed to decrease the processing time of our initial code. Thus, we used the bitwise shift operator to multiply our values by 256 (2^8). This way we could also neglect the image normalization process in the beginning (this of course does not help much in the accelerated functions but helped reducing the global execution time). (Run time reduced to 3.8s+).

HLS compliant: All above had to be done with two main constraints: use deterministic loops and avoid using pointers. These limitations (among many others, e.g. limited structure usage) guided us to keep our solution high-level synthesis (HLS) compliant. HLS is an automated design process to create register transfer level (RTL) design (in our case in form of VHDL, but it can output e.g. Verilog) based on a higher level language – in our case C. In order to achieve an efficient hardware implementation, we applied to the fixed point implementation different type of pragmas for:

- Kernel optimization (resource utilization): HLS resource, that implements a variable in RTL using the specified cores (e.g. MulNs for multiplication, or balancing between BRAM, LUT resources)
- Induce parallelization: HLS pipeline for allowing concurrent execution operation, HLS unroll to divide loop operations into independent ones, HLS array_partition to increase the amount of read and write ports for the storage

Additionally, extra dependencies (hdf5 library) had to be eliminated, therefore we created a weights.h file that contains weights and biases hard-coded.

Changes for compilation

After successful synthetization with pragmas we still encountered some errors while moving logic to hardware (HW version with pragmas). These errors helped us to understand more about HLS design.

Error #1: No virtual to physical mapping found

In conv1 function the input could not be partitioned (HLS array_partition), because it is connected to the main top level function (lenet_cnn), thus, to the port of the main accelerator (we used a single accelerator rooting from lenet_cnn). Introducing a temporary array solved the issue and we were able to achieve lower latency.

Error #2: Estimated clock period exceeds the target

Maybe this error showcased the best way how HLS resource pragma can help utilizing different cores. This occurred first with the conv1 layer (at the convolution operation, the multiplication), we introduced the following line: HLS RESOURCE variable=conv_px_sum core=MulnS latency=2, issue resolved.

Error #3: Design failed to meet timing

Maybe the trickiest message we had to solve, and the most change we had to apply to our logic. It occurred after introducing pragmas to our pooling layer. After checking the timing reports, the logic level for the pool1 function was 17 (for the rest it was around 9-11). We restructured our solution and finally we added some crucial pragmas to the block (mostly the pipeline was needed to reduce the latency).

Executable versions

Finally, we created three versions to compare the performance of our solution: SW, HW, HW + pragma. The first SW refers to the scenario, where the code is running on the device processors only. The two latter refer to the hardware accelerated versions (lenet_cnn as the top level HW function using a single accelerator), our expectation was that the last is going to result in the lowest clock cycle number, shortest global execution time and the highest utilization of FPGA resources.

Timing results

As expected, the accelerated version with applied pragmas were the most efficient from time perspective. The minimum number of clock cycle measure is 24 times less than the HW version without pragmas and is 3 times faster at the global execution time. Another substantial improvement is the reduction of clock cycles (there is a major difference between the minimum and the maximum values).

For the clock cycle measurements, we used sds_clock_counter function from sds_lib.h. The global time is calculated with the results of the function gettimeofday.

Version	Global time	Local min (clk)	Local max (clk)
SW	389s	7 413 302	16 111 340
HW	908s	43 946 334	48 133 712
HW + pragma	310s	1 854 234	5 928 854

Figure 2. Timing measurement comparison – reliable min and max values, tricky avg values

This is the consequence of the reduced latency shown by vivado hls: from 6 506 451 to 253 761 (**26X improvement**). The heaviest function was the conv2 (4.5 million latency) but pipelining those loops with more than 50 000 iterations (without counting the convolution itself) and combining this solution with array partitioning and different resource usage techniques resulted in great advancement. (For full report, please refer to the folder synthesis_results in the gitlab repository)

Latency		Interval		Type
min	max	min	max	
253761	253761	253761	253761	none

Figure 3. Latency after pragma addition

Energy consumption

For the accelerated versions, the total on-chip (thermal) power evidently increased from **0.183W (HW)** to **0.417W (HW + pragma)**, since it is the power consumed internally within the FPGA, and our main goal was to add pragmas to reduce latency by increasing resource usage. This consumption can be divided into dynamic and static power (power required for the FPGA to operate normally), and in the accelerated version – as expected - the power of the user design (dynamic) has significantly increased. The energy consumption of the **SW** (CPU) version is much higher, **1.707W.**

Power by itself does not give us much insight on the energy efficiency, for that we need to take the execution time into account as well. In the following two figures we can see the calculated energy for each solution based on the total on-chip power (Figure 4.), the used dynamic power (Figure 5.) and the above-mentioned global execution time:

Version	Global time (s)	Total Power (W)	Energy
SW	389	1.707	664.02
HW	908	0.183	166.16
HW + pragma	310	0.417	129.27

Figure 4. Energy measures (total on-chip power)

Version	Global time (s)	Dynamic Power (W)	Energy
SW	389	1.56	606.84
HW	908	0.075	68.1
HW + pragma	310	0.305	94.55

Figure 5. Energy measures (dynamic power)

In the end we can see that the most performant (fastest) version is the accelerated, parallelized one (HW + pragma), and it is also the most energy efficient version considering the total on-chip power. Comparatively, we increased the energy efficiency by around a **factor of 6** with the parallel FPGA solution compared to the version ran on the processor. If we check the dynamic power, the accelerated sequential solution consumes the least (compared to the SW version, it has an increased efficiency of **factor 10**).

Overall, we can conclude, that the hardware accelerated versions use generally way less energy (**6-10 times less**), and from the execution time's perspective the software and the parallel FPGA solution are more performant (**3 times**) than the sequential HW accelerated one. If we combine the two aspects (energy efficiency and the execution time), the **final parallel, hardware accelerated version (HW + pragma) can be announced as an absolute winner.**

For the detailed power usage and expected resource utilization, please refer to the section **Power consumption, resource utilization** in the appendix.

Conclusion

The project offered us various topics to understand through different tools: we got a glimpse into CNNs with python and C, hardware acceleration via HLS with vivado hls and SDSoC. Our results show that we not only achieved a significant improvement in timeliness but also save energy with our final solution.Due to the nature of the CNN layers, we used many loops in our solutions which causes high latency because of the iterations. We focused mainly on the pragma techniques related directly to parallelization and reached a satisfactory latency.

A possible improvement would be to try different pragmas as well to increase read and write operations through the whole logic, or try different parallelization techniques (e.g. combining HLS pipeline and HLS dataflow instructions). Another area to explore could be different combination or BRAM vs. LUT usage (e.g. HLS array_reshape could reduce BRAM consumption while sustaining parallel data access – similarly to HLS array_partition).

Furthermore, since most of the latency was caused by conv1, conv2 and fc1 functions, we focused mainly on those three logics with our pragmas (we reached 25-26-40X speedup). Although the overall execution time would not decrease significantly, a possible improvement would involve more optimization techniques in the pooling layers (mainly in pool1, due to the number of iterations).

Appendix

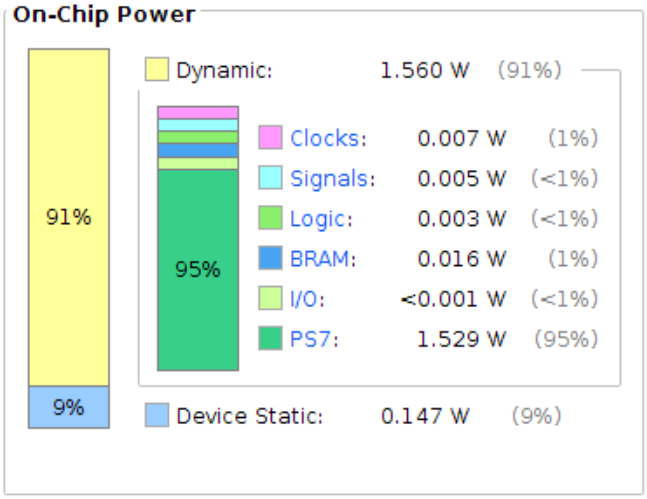
Power consumption, resource utilization

Power consumption (SW – HW – HW + pragma)

Power analysis from Implemented netlist. Activity derived from constraints files, simulation files or vectorless analysis.

Total On-Chip Power:	**1.707 W**
Design Power Budget:	**Not Specified**
Power Budget Margin:	**N/A**
Junction Temperature:	**44,7°C**
Thermal Margin:	40,3°C (3,4 W)
Effective ϑJA:	11,5°C/W
Power supplied to off-chip devices:	0 W
Confidence level:	Medium

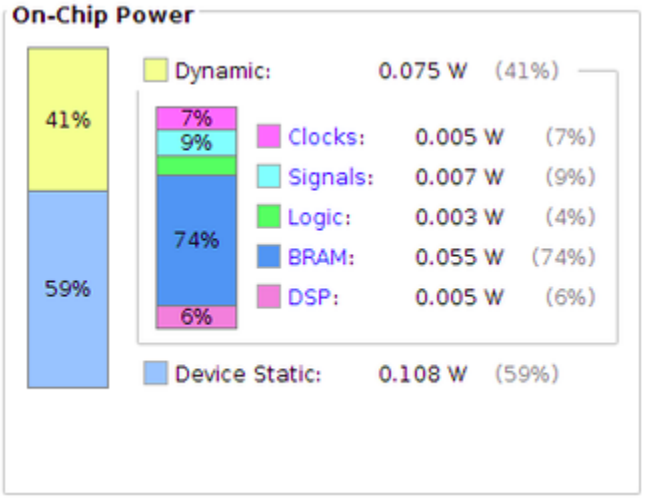
Launch Power Constraint Advisor to find and fix invalid switching activity



Power analysis from Implemented netlist. Activity derived from constraints files, simulation files or vectorless analysis.

Total On-Chip Power:	**0.183 W**
Design Power Budget:	**Not Specified**
Power Budget Margin:	**N/A**
Junction Temperature:	**27,1°C**
Thermal Margin:	57,9°C (4,8 W)
Effective ϑJA:	11,5°C/W
Power supplied to off-chip devices:	0 W
Confidence level:	Medium

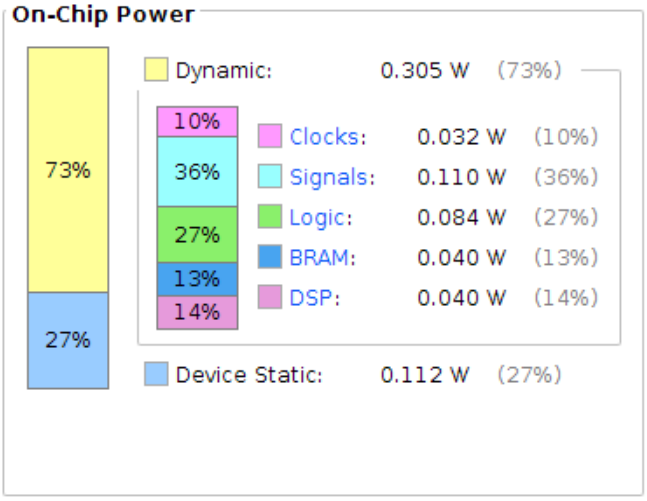
Launch Power Constraint Advisor to find and fix invalid switching activity



Power analysis from Implemented netlist. Activity derived from constraints files, simulation files or vectorless analysis.

Total On-Chip Power:	**0.417 W**
Design Power Budget:	**Not Specified**
Power Budget Margin:	**N/A**
Junction Temperature:	**29.8°C**
Thermal Margin:	55.2°C (4.6 W)
Effective ϑJA:	11.5°C/W
Power supplied to off-chip devices:	0 W
Confidence level:	Medium

Launch Power Constraint Advisor to find and fix invalid switching activity



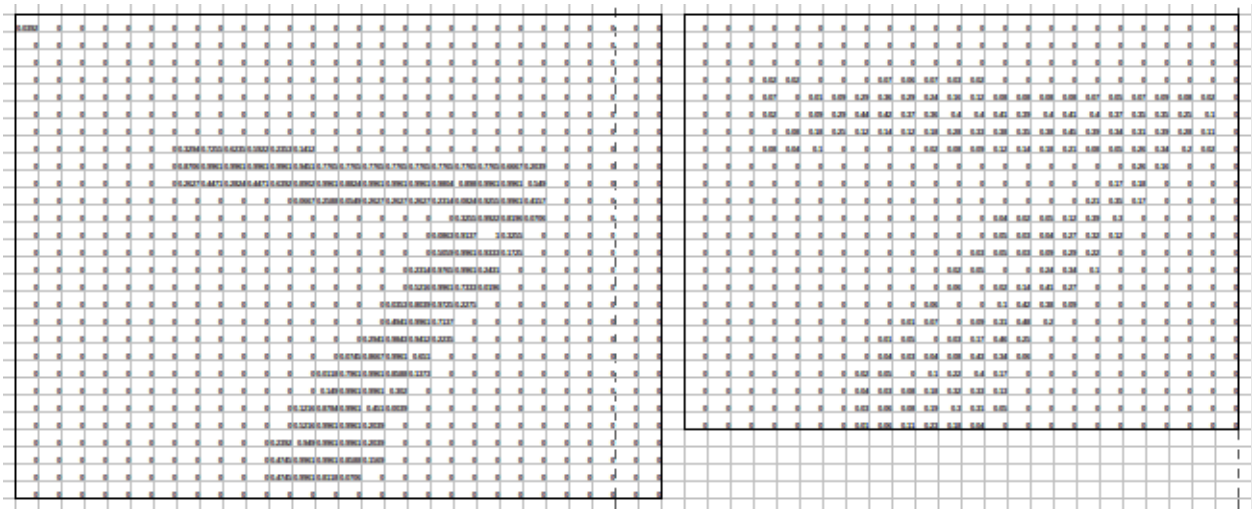
Resource utilization (without pragmas on the left, without on the right)

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	1	-	-
Expression	-	-	0	97
FIFO	-	-	-	-
Instance	117	3	1357	3107
Memory	28	-	5	1
Multiplexer	-	-	-	314
Register	-	-	128	-
Total	145	4	1490	3519
Available	280	220	106400	53200
Utilization (%)	51	1	1	6

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	1	-	-
Expression	-	-	0	97
FIFO	-	-	-	-
Instance	133	41	15304	24792
Memory	2	-	1077	4141
Multiplexer	-	-	-	2207
Register	-	-	128	-
Total	135	42	16509	31237
Available	280	220	106400	53200
Utilization (%)	48	19	15	58

LeNet-5 visualization

conv1 input, convolution result (t10k-images-idx3-ubyte[00000])



pool1 output, conv2 output (t10k-images-idx3-ubyte[00000])

0	0	0	0	0	0	0	0	0	0	0	0
0	0.022	0.024	0	0.071	0.07	0.017	0	0	0	0	0
0	0.066	0.091	0.436	0.424	0.403	0.406	0.399	0.412	0.366	0.352	0.098
0	0.079	0.184	0.245	0.138	0.277	0.376	0.381	0.454	0.34	0.39	0.112
0	0	0	0	0	0	0	0	0	0.264	0.158	0
0	0	0	0	0	0	0.04	0.051	0.393	0.352	0	0
0	0	0	0	0	0	0.05	0.085	0.325	0.117	0	0
0	0	0	0	0	0.059	0.052	0.414	0.339	0	0	0
0	0	0	0	0.01	0.068	0.305	0.478	0.093	0	0	0
0	0	0	0	0.053	0.076	0.456	0.254	0	0	0	0
0	0	0	0.037	0.079	0.322	0.395	0	0	0	0	0
0	0	0	0.028	0.105	0.295	0.307	0	0	0	0	0

0	0	0	0	0	0	0	0
3.823	4.882	4.927	4.39	3.69	2.884	2.35	2.281
3.871	4.465	4.337	3.832	2.836	2.306	2.517	2.442
1.279	1.165	0.56	0	0	0	1.537	2.15
0.218	0	0	0	0	1.075	3.21	2.295
0	0	0	0	0.296	3.179	3.822	1.647
0	0	0	0	2.317	4.379	3	0.383
0	0	0	0.865	4.145	4.059	1.497	0

Bias 40 [0] -0.01

Results on the ZedBoard

[1] SW [2] HW [3] HW+pragma [4] after cached

[1]

```
Opening labels file

Processing
TOTAL PROCESSING TIME (gettimeofday): 389.153764 s


Errors : 201 / 10000

Success rate = 97.989998%

Thw_min = 7413302 cpu cycles     Thw_max = 16111340 cpu cycles   Thw_avg = 7435

root@zed:/media#
```

[2]

```
Opening labels file

Processing
TOTAL PROCESSING TIME (gettimeofday): 908.224543 s


Errors : 201 / 10000

Success rate = 97.989998%

Thw_min = 43946334 cpu cycles    Thw_max = 48133712 cpu cycles   Thw_avg = 4396

root@zed:/media#
```

[3]

```
Opening labels file

Processing
TOTAL PROCESSING TIME (gettimeofday): 310.408472 s


Errors : 201 / 10000

Success rate = 97.989998%

Thw_min = 1854234 cpu cycles     Thw_max = 5928854 cpu cycles    Thw_avg = 1872

root@zed:/media#
```

[4]

```
Opening labels file

Processing
TOTAL PROCESSING TIME (gettimeofday): 32.295618 s


Errors : 201 / 10000

Success rate = 97.989998%

Thw_min = 1844082 cpu cycles     Thw_max = 6150868 cpu cycles    Thw_avg = 1849

root@zed:/media#
```

References

[1] LeNet-5 – A Classic CNN Architecture, https://engmrk.com/lenet-5-a-classic-cnn-architecture/

[2] LeNet – Convolutional Neural Network in Python,

https://www.pyimagesearch.com/2016/08/01/lenet-convolutional-neural-network-in-python/

[3] Convolutional Neural Networks (CNNs / ConvNets), https://cs231n.github.io/convolutional-Networks/

[4] Convolutional Neural Networks, https://leonardoaraujosantos.gitbooks.io/artificial-inteligence/content/convolutional_neural_networks.html

[5] Looking inside neural nets, https://ml4a.github.io/ml4a/looking_inside_neural_nets/

[6] Ubuntu 18.04: Install TensorFlow and Keras for Deep Learning, https://www.pyimagesearch.com/2019/01/30/ubuntu-18-04-install-tensorflow-and-keras-for-deep-learning/

[7] Coding Considerations (Xilinx), http://home.mit.bme.hu/~szanto/education/vimima15/heterogen_vivado_hls_6.pdf

[8] HLS pragmas (xilinx), https://www.xilinx.com/html_docs/xilinx2019_1/sdaccel_doc/hls-pragmas-okr1504034364623.html

[9] Xilinx – Unable to schedule 'load' operation, https://forums.xilinx.com/t5/High-Level-Synthesis-HLS/Unable-to-schedule-load-operation/td-p/937007

[10] Karparthy - ConvNetJS MNIST demo, https://cs.stanford.edu/people/karpathy/convnetjs/demo/mnist.html

[11] Keras – Layer Activation functions, https://keras.io/api/layers/activations/

[12] Corvell – Fixed point types, analysis of algorithms, https://www.csl.cornell.edu/courses/ece5775/pdf/lecture04.pdf

[13] GitLab repository – team 11, https://gitlab.polytech.unice.fr/nn012527/hls_ia_team11

[14] https://www.xilinx.com/products/design-tools/software-zone/sdsoc.html

[15] https://www.xilinx.com/support/documentation/sw_manuals/xilinx2012_4/ug907-vivado-power-analysis-optimization.pdf

[16] https://reference.digilentinc.com/_media/zedboard:zedboard_ug.pdf