National Technical University of Athens
Electrical and Computer Engineering

Embedded Systems

4th Lab

HIGH-LEVEL SYNTHESIS WORK ON FPGA

Nanos Georgios

nanosgiwrgos1997@gmail.com

The purpose of the exercise is to make a study around the FPGA programming with High Level Synthesis (HLS). optimization and acceleration of C code to eventually run on the hardware of the Xilinx Zybo FPGA. The application to be studied for acceleration will be related to neural networks and in particular Generative Adversarial Networks (GANs). Specifically, the GAN application provided ready relates to reconstruction of incomplete images from handwritten digits. The final goal is to speed up this algorithm with respect to the SW implementation but and to measure the reconstruction quality of the images. The implementation concerns the reconstruction of images from the MNIST dataset (28x28 handwritten grayscale figures). In particular, we have the generator neural model from GAN where we are given its C implementation together with the trained weights. The implementation takes as input the top half of the numbers and calls the neural to generate/predict the remaining half of image. After execution, the average execution time per image will be shown in cycles for SW and HW and the speed-up. The goal is to maximize the speed-up with HLS optimizations in the HW function. The output.txt that generates the application is then processed via Jupyter to display the numbers.

The Source files of the Application are the following:

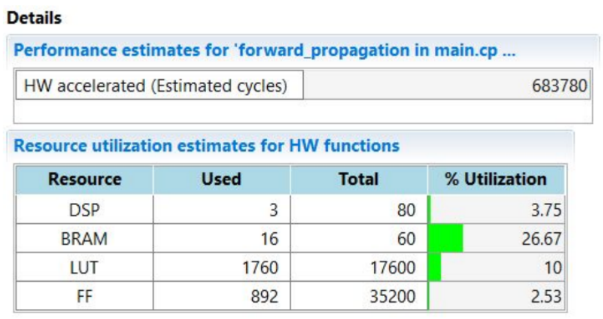
- main.cpp. Main CPU: CPU calls the Generator model in SW and HW and measures the performance.
- network.cpp: It is the core code of the application i.e. the application's It is the core of the application, which is the Generator. This should be optimized with HLS to run on the HW (only the forward_propagation function needs optimizations).
- weight_definitions.h: Contains the trained weights of the neural network. generator.
- tanh.h: Stores the pre-computed values of the mathematical function Tanh for HW.
- network.h: Contains various Generator arguments (datatypes, loop limits, etc.).
- data.txt: Contains the input dataset (the top half of the images).

It is required to be in the executable folder when running on the ARM and also in plot_output.ipynb. Through the SDSoC tool we develop the application for the embedded Zybo FPGA. Specifically, SDSoC provides an Eclipse type environment where we write C/C++ code and develop a accelerator that runs on FPGA using HLS optimizations. we can do performance and resource estimation, build the bitstream of the hardware and eventually the sd boot card where the system (the generated files inside the sd_card include the executable (.elf), the bitstream and the system's petalinux operating system).

After making a Xilinx SDx project according to the above configuration (zybo, linux, debug mode, 100Mhz) and import the source files given to us, mark the function we want to implement in the HW. In this case it is forward_propagation which at the end will have all the HLS optimizations we need to add. Finally, we build the project each time to implement what we want.

Issue 1: Performance and resources measurement

The forward_propagation function is the C code that runs the neural, specifically the generator. It is several layers that run the one after the other (dense->relu->dense->relu->dense->tanh) in order to generate the bottom half of the image. Basically they are poles matrix-vector which is a CPU-slow, multi-operation process. A) First we tried to set it as a HW function and made estimate performance without performing any optimization. We recorded the report showing the estimated resources and cycles of the hardware function.


Details

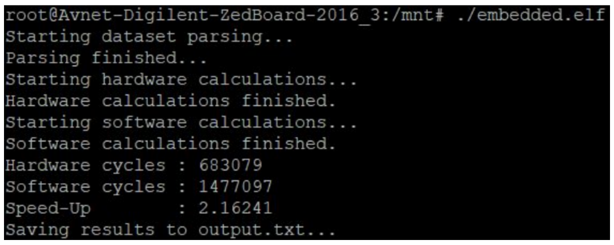
Performance estimates for 'forward_propagation in main.cp ...

HW accelerated (Estimated cycles)	683780

Resource utilization estimates for HW functions

Resource	Used	Total	% Utilization
DSP	3	80	3.75
BRAM	16	60	26.67
LUT	1760	17600	10
FF	892	35200	2.53

B) We then tried to build the sd_card with the bitstream and pass it to the zybo sd. (We ignored possible warnings during the compilation (unused labels, etc.)). After passing the data.txt, we did rebooted and ran the application on the board. We noticed, as shown from the screenshot below, that it agrees with the estimation.The speed-up compared to SW execution on ARM is 2.16241.

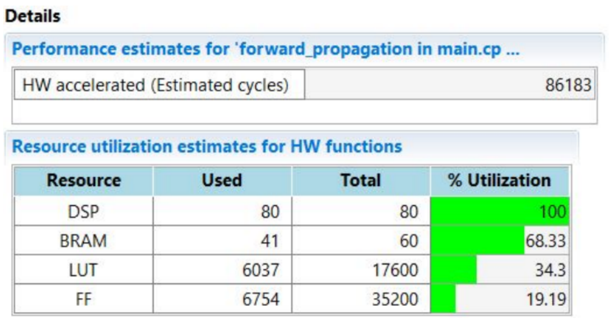

```
root@Avnet-Digilent-ZedBoard-2016_3:/mnt# ./embedded.elf
Starting dataset parsing...
Parsing finished...
Starting hardware calculations...
Hardware calculations finished.
Starting software calculations...
Software calculations finished.
Hardware cycles : 683079
Software cycles : 1477097
Speed-Up        : 2.16241
Saving results to output.txt...
```

C) At this point design space exploration was carried out in order to optimizations were found and the algorithm was significantly accelerated. We tried several HLS pragmas and saw what the estimation comes out. Pipeline: Loop pipelining is the most important optimization in HLS. It allows a new iteration to start processing before the loop is complete. previous iteration. In our example, we noticed that we did not have internal dependencies, so we defined pipelining on all loops.

```
read_input:
for (int i=0; i<N1; i++)
{
#pragma HLS pipeline
    xbuf[i] = x[i];
}
// Layer 1
layer_1:
for(int i=0; i<N1; i++)
{
#pragma HLS pipeline
    for(int j=0; j<M1; j++)
    { ...

layer_1_act:
for(int i=0; i<M1; i++)
{
#pragma HLS pipeline
    layer_1_out[i] = ReLU(layer_1_out[i]);
}
// Layer 2
layer_2:
for(int i=0; i<M2; i++)
{
#pragma HLS pipeline
    l_quantized_type result = 0;
    for(int j=0; j<N2; j++)
    { ...
// Layer 3
layer_3:
for(int i=0; i<M3; i++)
{
#pragma HLS pipeline
    l_quantized_type result = 0;
    for(int j=0; j<N3; j++)
    { ...
```

Unroll Factor: We added the pragma unroll to the inner double loops in order to to achieve simultaneous computation of the individual sums that We added the additional unroll factor to add the multiplication of all the individual elements in the same row (Layer1-W1) or column (Layer2-W2, Layer3-W3). In the Unroll command, a factor equal to 30 in layer 1, 2 and 50 in layer 3. These values were chosen according to the total iterations of the inner loops.

```
// Layer 1
layer_1:
for(int i=0; i<N1; i++)
{
#pragma HLS pipeline
    for(int j=0; j<M1; j++)
    {
    #pragma HLS unroll factor=30
        l_quantized_type last = (i==0) ? (l_quantized_type) 0 : layer_1_out[j
    ];
        quantized_type term = xbuf[i] * W1[i][j];
        layer_1_out[j] = last + term;
    }
}

// Layer 2
layer_2:
for(int i=0; i<M2; i++)
{
#pragma HLS pipeline
    l_quantized_type result = 0;
    for(int j=0; j<N2; j++)
    {
    #pragma HLS unroll factor=30
        l_quantized_type term = layer_1_out[j] * W2[j][i];
        result += term;
    }
    layer_2_out[i] = ReLU(result);
}

// Layer 3
layer_3:
for(int i=0; i<M3; i++)
{
#pragma HLS pipeline
    l_quantized_type result = 0;
    for(int j=0; j<N3; j++)
    {
    #pragma HLS unroll factor=50
        l_quantized_type term = layer_2_out[j] * W3[j][i];
        result += term;
    }
    y[i] = tanh(result).to_float();
}
```


Details

Performance estimates for 'forward_propagation in main.cp ...

HW accelerated (Estimated cycles)	86183

Resource utilization estimates for HW functions

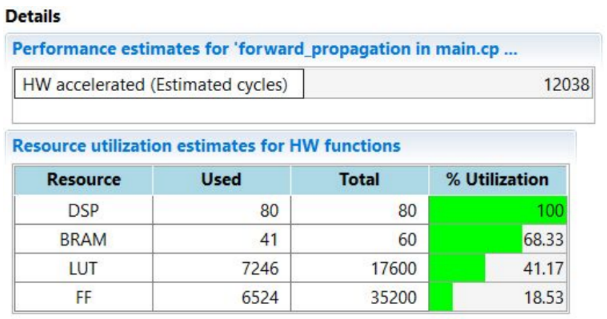
Resource	Used	Total	% Utilization
DSP	80	80	100
BRAM	41	60	68.33
LUT	6037	17600	34.3
FF	6754	35200	19.19

Array Partition: Because the memory has a limited number of read ports and write ports, we divided the original array into smaller ones. This effectively increases the number of the array. load/store ports and thus the memory bandwidth is improved. We experimented with different values for the types and the factor in the pragmas. array_partition and came up with the following. The final values that resulting are essentially divisors of the original array size. For the two-dimensional ones, we chose to make partition the dimension that traversed by the inner double loop.

```
#pragma HLS array_partition variable=layer_1_out block factor=15 dim 1
#pragma HLS array_partition variable=layer_2_out block factor=25 dim 1
#pragma HLS array_partition variable=W1 block factor=15 dim 2
#pragma HLS array_partition variable=W2 block factor=15 dim 1
#pragma HLS array_partition variable=W3 block factor=25 dim 1
```

Controlling resources: To limit the number of functions that will be implemented in the hardware, the pragma HLS allocation command is used. We note that at the first level we have 30 neurons. In each neuron, we use the pragmatic HLS algorithm. the function ReLU is called, so in total it is called 30 times. Also, the Tanh function is called 1 time in the last level. Based on this we define the following limits:

```
#pragma HLS allocation instances=tanh limit=1 function
#pragma HLS allocation instances=ReLU limit=30 function
```


Details

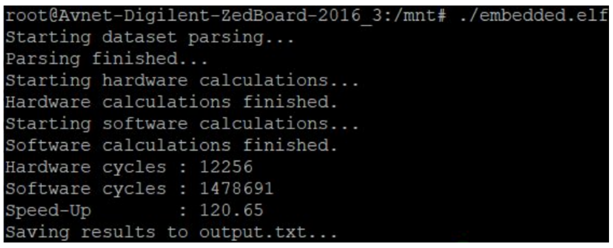
Performance estimates for 'forward_propagation in main.cp ...

HW accelerated (Estimated cycles)	12038

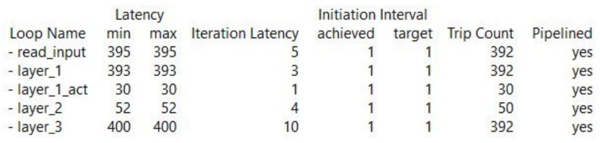
Resource utilization estimates for HW functions

Resource	Used	Total	% Utilization
DSP	80	80	100
BRAM	41	60	68.33
LUT	7246	17600	41.17
FF	6524	35200	18.53

We produced the sd with the bitstream and ran the application on the board. We re-recorded cycles and speed-up from the board. We observed again, as shown in the screenshot below, that it matches the estimation.

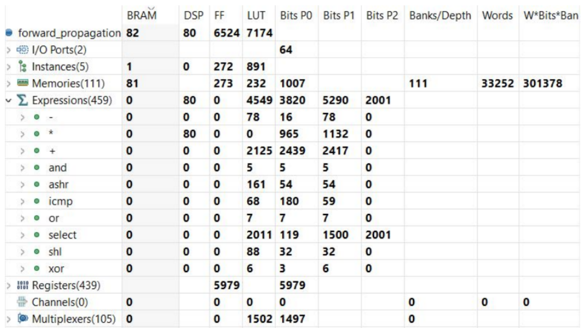

```
root@Avnet-Digilent-ZedBoard-2016_3:/mnt# ./embedded.elf
Starting dataset parsing...
Parsing finished...
Starting hardware calculations...
Hardware calculations finished.
Starting software calculations...
Software calculations finished.
Hardware cycles : 12256
Software cycles : 1478691
Speed-Up        : 120.65
Saving results to output.txt...
```

Comparing this implementation (optimized), with the original (unoptimized), we notice that the Speed-Up from 2.16 has reached 120.65, i.e. over over 55 times faster compared to the original. D) We also saw for our optimized implementation in SDSoC the HLS report generated. We recorded the latency details for each loop (Latency -> Detail -> Loop).



	Latency			Initiation Interval			
Loop Name	min	max	Iteration Latency	achieved	target	Trip Count	Pipelined
- read_input	395	395	5	1	1	392	yes
- layer_1	393	393	3	1	1	392	yes
- layer_1_act	30	30	1	1	1	30	yes
- layer_2	52	52	4	1	1	50	yes
- layer_3	400	400	10	1	1	392	yes

We notice that the layer with the highest latency is layer_3. Also the our design is fully pipelined. After heading to the Resource profile tab of the HLS report, we listed the types of mathematical expressions in the design.

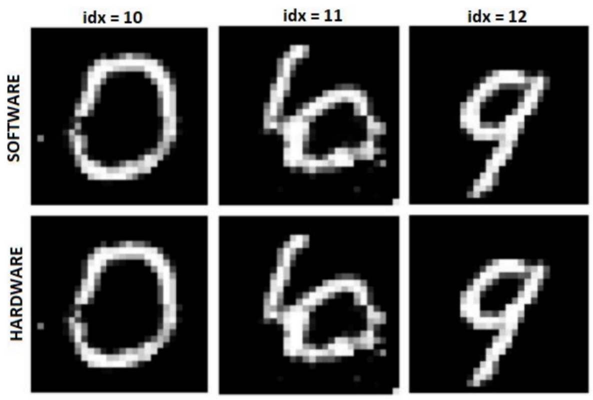


	BRAM	DSP	FF	LUT	Bits P0	Bits P1	Bits P2	Banks/Depth	Words	W*Bits*Ban
forward_propagation	82	80	6524	7174						
I/O Ports(2)					64					
Instances(5)	1	0	272	891						
Memories(111)	81		273	232	1007			111	33252	301378
Expressions(459)	0	80	0	4549	3820	5290	2001			
-	0	0	0	78	16	78	0			
*	0	80	0	0	965	1132	0			
+	0	0	0	2125	2439	2417	0			
and	0	0	0	5	5	5	0			
ashr	0	0	0	161	54	54	0			
icmp	0	0	0	68	180	59	0			
or	0	0	0	7	7	7	0			
select	0	0	0	2011	119	1500	2001			
shl	0	0	0	88	32	32	0			
xor	0	0	0	6	3	6	0			
Registers(439)			5979		5979					
Channels(0)	0		0	0	0			0	0	0
Multiplexers(105)	0		0	1502	1497			0		

The expression that most DSPs need is multiplication.

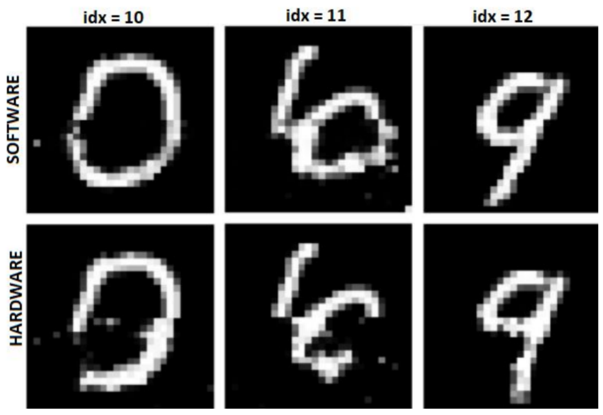
Issue 2: Quality measurement

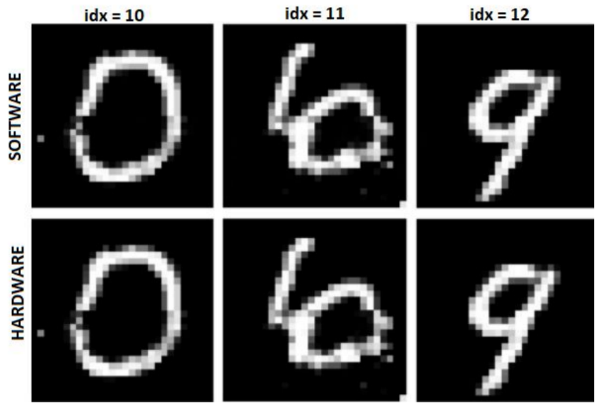
At this point we were asked to measure the quality of reconstruction of the images through the jupyter notebook we were given. Through this we made combine half of the images given as input to data.txt with half of the images from the output.txt produced by both SW and HW. A) The source files given to us for SDSoC run the algorithm of generator successfully. We transferred the output.txt that produced the executable from the board to our pc (no need to optimize the design). We ran the plot_output.ipynb from google collab online. We displayed the combined images from SW and HW for various numbers, specifically for idx: 10, 11, 12.



B) The implementation given uses datatypes that have 8-bit decimal precision. To change it, we first changed the BITS and BITS_EXP in the network.h file where BITS_EXP = 2.(BITS+2) we changed the pre-computed values of Tanh in the tanh.h file. We were given the computed values for various bits in the tanh file. We tried to create new designs with 4 and 10 bits and run them. We displayed the combined images resulting from the output again for idx: 10,11,12.

	4 bits			8 bits			10 bits		
	idx 10	idx 11	idx 12	idx 10	idx 11	idx 12	idx 10	idx 11	idx 12
mpe	255	249	255	16	17	13	5	5	4
psnr	14.05	14.63	13.52	42.68	42.56	47.06	54.08	52.55	53.76





We note that as the number of decimal precision bits increases, the more the number of bits the quality of image reconstruction improves. This is because with more bits we have more pre-computed values of Tanh and thus the quantization error is smaller. C) In the jupyter notebook the image reconstruction quality of the HW compared to SW. For the different bits we tested (4,8,10) we show max pixel error and Peak Signal-to-Noise Ratio (psnr).

Our preferred technique is PSNR for two reasons:

- If two images differ only in one pixel and in all other pixels are identical in all other respects, then Max Pixel Error will take the maximum value (255). Even though the images are almost identical. In contrast, the PSNR takes a value of takes all pixels into account, since the average value is calculated and thus achieves an overall estimate of the reconstruction quality.
- As can be seen from the third case (10 bits), it can for Max Pixel Error may be the same (integer value) for some images and no conclusion can be drawn. However, this would not occur in the PSNR due to the decimal representation it receives.

As can be seen from the above table, the bit precision (from 4 to 10) that is ideal is 10, as it achieves a lower MPE and a higher PSNR. This was obviously seen in the figures in question B, where we clearly had significantly better reconstruction quality. The negative with increasing the decimal precision bits is that it increases the requirements on memory, since the size of the tanh_vals table containing the pre-computed values of tanh.