

Implementation of a Binary CNN on FPGA with High-Level Synthesis Tools

RICCARDO ALBERTAZZI – UNIVERSITY OF BOLOGNA - 2018

PROJECT WORK OF “RECONFIGURABLE EMBEDDED SYSTEMS”
PROF. STEFANO MATTOCCIA

riccardo.albertazzi.it@gmail.com

<https://github.com/riccardoalbertazzi/bnn>

References

Reference paper about Binary Networks:

- **Binarized Neural Networks: Training Deep Neural Networks with Weights and Activations Constrained to +1 or -1**

<https://arxiv.org/abs/1602.02830>

Keras implementation of BinaryNet:

- https://github.com/DingKe/nn_playground

Other FPGA implementations:

- FINN: A Framework for Fast, Scalable Binarized Neural Network Inference
<https://arxiv.org/abs/1612.07119>
- Accelerating Binarized Convolutional Neural Networks with Software-Programmable FPGAs
<https://dl.acm.org/citation.cfm?id=3021741>

Neural Networks on FPGA

- Modern deep learning networks need to perform an enormous amount of floating point computation during both training and inference. A vast majority of these operations can be computed in parallel, and this is why GPUs are usually the target devices for these tasks. For very deep neural networks, multiple GPUs can be used in parallel.
- Unfortunately, a single GPU's power consumption can be about 200/300W and its weight about 1kg. These are not good specifications for embedded or mobile devices.
- This is way a lot of research has been done on implementing neural networks on power-friendly devices like FPGAs. Field Programmable Gate Arrays are small devices with little power consumption (about 1/2W) with reconfigurable hardware.
- FPGAs don't have the same speed nor the same amount of resources of a modern GPU, but an efficient implementation could obtain the same throughput (images/sec) of a GPU by using a significantly smaller amount of power.

Simplifying Neural Networks

- The basic operations for both convolutional and fully connected layers, that together hold almost all the computation of the network, are simple multiplications and additions/subtractions (MAC – Multiply and Accumulate).
- However, floating-point operations require a lot of resources compared to other possible solutions, such as fixed-point or integer solutions.
- In many papers researchers have shown that it's possible to train a network with simpler value types without significant accuracy loss.

BinaryNet

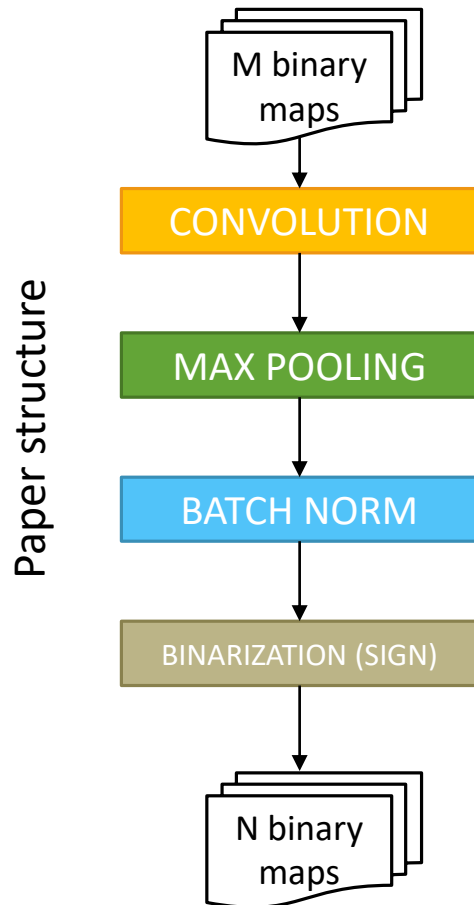
Courbarieaux et al. tried to reduce value types to the simplest and most efficient possible value: a binary value

- In their implementation they propose a fully binarized convolutional network, named BinaryNet, where both weights and activations (inputs and outputs of each layer) have only two values: -1 or 1

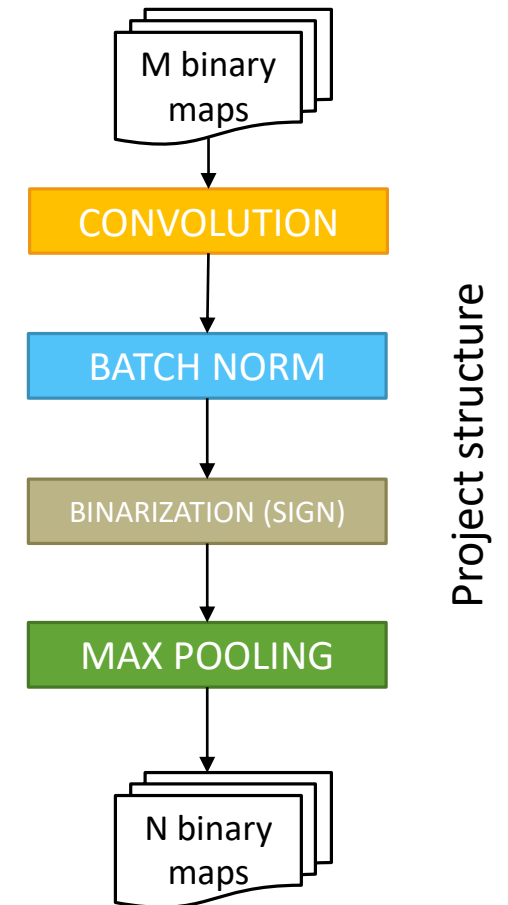
This is how the training process works, in summary:

- In order to deal with derivatives and gradient descent, floating-point weights are kept
- To achieve binarization and at the same time allow back propagation, a special function is applied to the weights: during the forward (inference) pass it behaves like a sign function; during the backward (weights update) pass it behaves like a derivable function
 - If the real sign function was used, it wouldn't be possible to propagate the error since its derivative is 0 everywhere
 - The same concept is applied to the output of each layer; in fact, if we apply a convolution with binary weights to a binary image, the result is an integer image. The output is then normalized (using canonical batch normalization) and thresholded using the sign function (again, a differential function). Same thing for fully connected layers (even though the last fully connected layer doesn't have a sign activation function).
- Bias variables for convolution and dense layers have been removed

Differences between original paper



- In the original paper the basic convolution structure consists of the following ordered layers: Convolution, Max Pooling, Batch Normalization, Activation Function (sign function)
- In this work the Max Pooling operation has been moved after the activation function (and before the next convolution)
 - By doing that, we let the Max Pooling operation work with binary values instead of integer values -> less resources (see later)



BinaryNet – Training and Testing

- We used and adapted some already available Keras code to train a network on the MNIST dataset
- MNIST: dataset of hand-written digits with 60k training images and 10k test images. Images are grayscale (one channel), 28x28 pixels. The number of classes is 10 (0 to 9 digits).
- Modern deep learning approaches reach 99% accuracy fairly easily
 - «If it works on MNIST, it may work for more interesting tasks»



BinaryNet – Training and Testing

This is the test network in summary:

- Conv 3x3, 32 filters
- Conv 3x3, 32 filters
- Max Pooling 2x2
- Conv 3x3, 64 filters
- Conv 3x3, 64 filters
- Max Pooling 2x2
- Dense, 128 outputs
- Dense, 10 outputs (classes)

Total number of weights: 468k.

About 85% of the total weights belong to the first fully connected layer.

Layer (type)	Output Shape	Param #
conv1 (BinaryConv2D)	(None, 28, 28, 32)	288
bn1 (BatchNormalization)	(None, 28, 28, 32)	128
act1 (Activation)	(None, 28, 28, 32)	0
conv2 (BinaryConv2D)	(None, 28, 28, 32)	9216
bn2 (BatchNormalization)	(None, 28, 28, 32)	128
act2 (Activation)	(None, 28, 28, 32)	0
pool2 (MaxPooling2D)	(None, 14, 14, 32)	0
conv3 (BinaryConv2D)	(None, 14, 14, 64)	18432
bn3 (BatchNormalization)	(None, 14, 14, 64)	256
act3 (Activation)	(None, 14, 14, 64)	0
conv4 (BinaryConv2D)	(None, 14, 14, 64)	36864
bn4 (BatchNormalization)	(None, 14, 14, 64)	256
act4 (Activation)	(None, 14, 14, 64)	0
pool4 (MaxPooling2D)	(None, 7, 7, 64)	0
flatten_2 (Flatten)	(None, 3136)	0
dense5 (BinaryDense)	(None, 128)	401408
bn5 (BatchNormalization)	(None, 128)	512
act5 (Activation)	(None, 128)	0
dense6 (BinaryDense)	(None, 10)	1280
bn6 (BatchNormalization)	(None, 10)	40
Total params: 468,808		

BinaryNet – Training and Testing

- With our test network we are able to achieve 95.5% accuracy
- In the original paper the squared hinge loss is used; in this work the network is trained with a more traditional crossentropy loss. The results are even better! However, we will need to compute a softmax operation (floating point op)
- Tests show that moving Max Pool operations after the binarization function results in slightly lower accuracy; however, it is considered to be a negligible loss compared to the advantages we get in terms of resource usage
- We also explored the possibility to remove the last batch normalization layer (motivations are explained in the next slides)
- With bigger networks (e.g. 1024 output neurons in the first dense layer) we can increase the accuracy to 97%.

MAX POOL AFTER ACTIVATION

MNIST (20 epochs)	Loss=hinge	Loss=crossentropy
w/ last batch norm	93%	95.5%
w/o last batch norm	81%	92.5%

MAX POOL BEFORE ACTIVATION

MNIST (20 epochs)	Loss=hinge	Loss=crossentropy
w/ last batch norm	94.6%	95.8%
w/o last batch norm	85%	92.9%

BinaryNet – Training and Testing

- We went from 32-bit float multiplications to a 1-bit XNOR operations, and from storing 32-bit weights to 1 bit weights; this means, for an FPGA, an enormous saving of resources!
- When targeting a low-end FPGA with a limited number of resources, it makes the difference between implementing a single layer or an entire network
- Also, since the memory requirements have been reduced by a factor of 32, it becomes feasible to store all the binary weights in the on-chip BRAMs, instead of loading them from an off-chip memory (such as an external RAM) each time -> faster computation

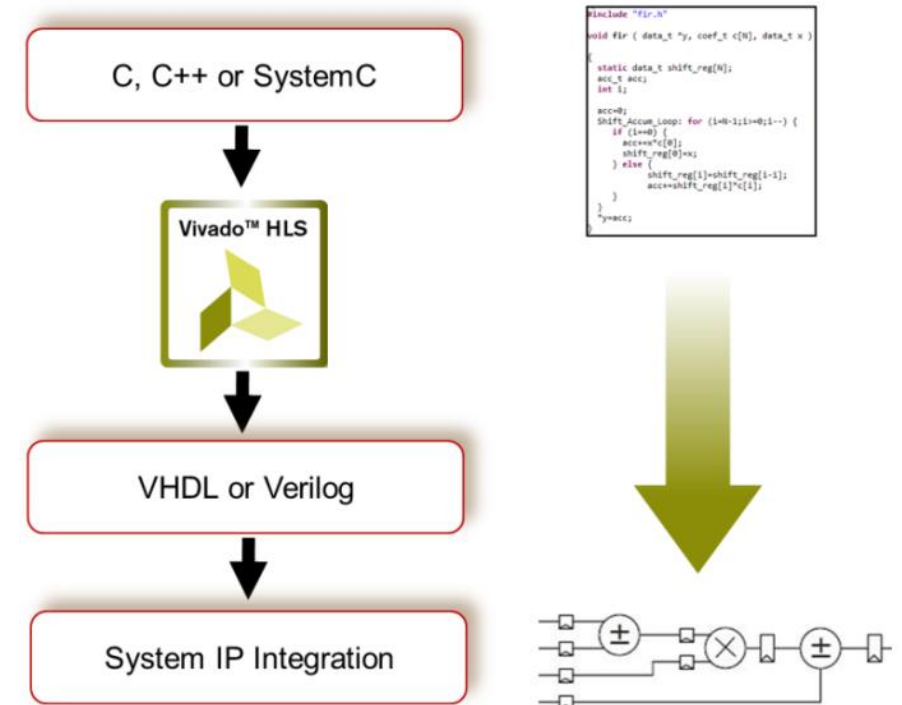
BinaryNet - Implementation

BinaryNet Implementation

- First we have shown that it is possible to train a network with binary values and activations
- Now we are going to synthesize it in hardware!
- Our target architecture is the Xilinx Zynq processing unit Z-7020, a low-end chip that contains two main components: a Processing System (PS), with a dual-core ARM CPU, a DDR memory and other peripherals (Ethernet, USB, SDCard, ...), and a Programmable Logic (PL), the FPGA itself, that can communicate with the PS part.
- The PL contains 106400 Flip-Flops, 53200 LUTs (core for every type of operation that we can synthesize) and 220 BRAMs (on-chip memory) with 18kb each

BinaryNet Implementation

- For this project we are going to use Xilinx's High-Level Synthesis Tool, Vivado HLS, that allows us to write C++ code that will be translated into RTL code (VHDL, Verilog); the major benefits of using Vivado HLS are its pragmas, that allow the programmer to control resource usage, timing requirements and the architecture implementation
- The most important pragmas allow for example:
 - To infer pipeling into our modules
 - To unroll loops and achieve parallelism
 - To reshape arrays in order to maximize memory usage
 - To control input and output ports via standard protocols (AXI, AXI-Stream)
- By exploring different configurations of pragmas we can evaluate multiple architectures and find the optimal one for our purposes (e.g. trade-off timing/resources)



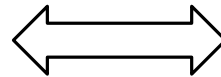
BinaryNet Implementation

- In this project we are going to implement the core modules that allow to perform the inference step (training is still assumed to be performed on GPUs):
 - Fully Connected (Dense) layers
 - Convolutional layers
 - Max pooling layers
 - Padding layers
- Other components will be mentioned or described in general terms

BinaryNet – Considerations for FPGA implementation

- BinaryNet values can be -1 or 1, but in hardware we want to store every value in just one bit, meaning that -1 will be represented by a 0. This requires to change the internal basic operations:
 - How can we express a sign multiplication by using zeros and ones?
 - XNOR (equivalence gate)

INPUT VALUE	WEIGHT VALUE	PRODUCT
1	1	1
1	-1	-1
-1	1	-1
-1	-1	1



INPUT VALUE	WEIGHT VALUE	XNOR
1	1	1
1	0	0
0	1	0
0	0	1

- The XNOR behaves like a sign multiplication, but the result is still different (0 != -1). This means that accumulating 0s and 1s will provide a different result than accumulating -1s and 1s. However, we can overcome this problems by simply employing different threshold values (it will be explained later)

Weights Loading Architecture

Weights Loading

- Since this work deals with binary weights, it is feasible to save all the network's weights in the FPGA resources:
 - On our target Zynq FPGA there's a total of 5 Mbit on-chip BRAMs, so we could implement a network with 5 million weights
 - In this work we assume that all weights will be saved on the FPGA resources (BRAMs or LUTs, depending on the number of weights and on the required degree of parallelism)
- We have to consider the loading phase of the network, where each layer will be loaded with the required weights
 - Before sending them to the FPGA layers, weights will be saved on the DDR memory of the Zynq system
 - The CPU needs to communicate with the FPGA in order to send the weights

Weights loading architecture (the one used in this work):

- Every hw module that requires weights has one 64-bit AXI-Stream input channel to read the weights and one interrupt, controlled by AXI-Lite interface, that is asserted by the CPU when new weights have to be loaded
- One bridge module will be responsible of reading the weights on the off-chip DDR (using an AXI-Master interface connected to the HP ports) and send them to the designated layer using the AXI-Stream protocol

In an alternative architecture we could equip each layer with an AXI-Master that communicates directly with the DDR memory.

Weights Loading

Advantages of the first approach over the alternative architecture:

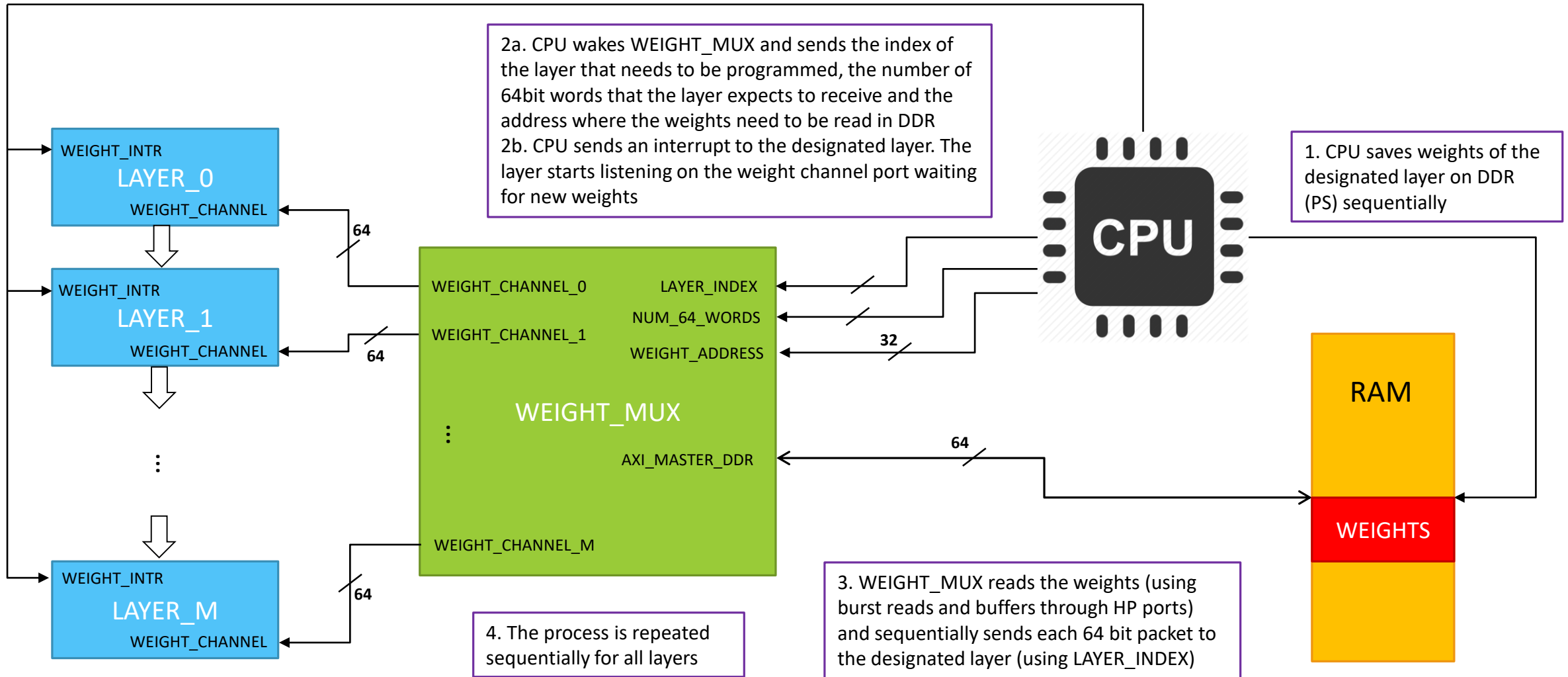
- AXI-Stream is very lightweight in term of resources; on the opposite, if every module used an AXI-Master interface we would use a lot of resources just for reading the weights / buffering data

Disadvantages of the first approach over the alternative architecture:

- With one module that needs to read and send the weights to every layer, we will need to program each layer sequentially; this would become a problem if we had to change the weights frequently. We could solve the problem anyways by employing more than one instance of the described bridge module; each instance would be responsible to communicate with a subset of the network layers (there could even be one bridge module for each layer!)

In this work the first approach is employed (see next slide)

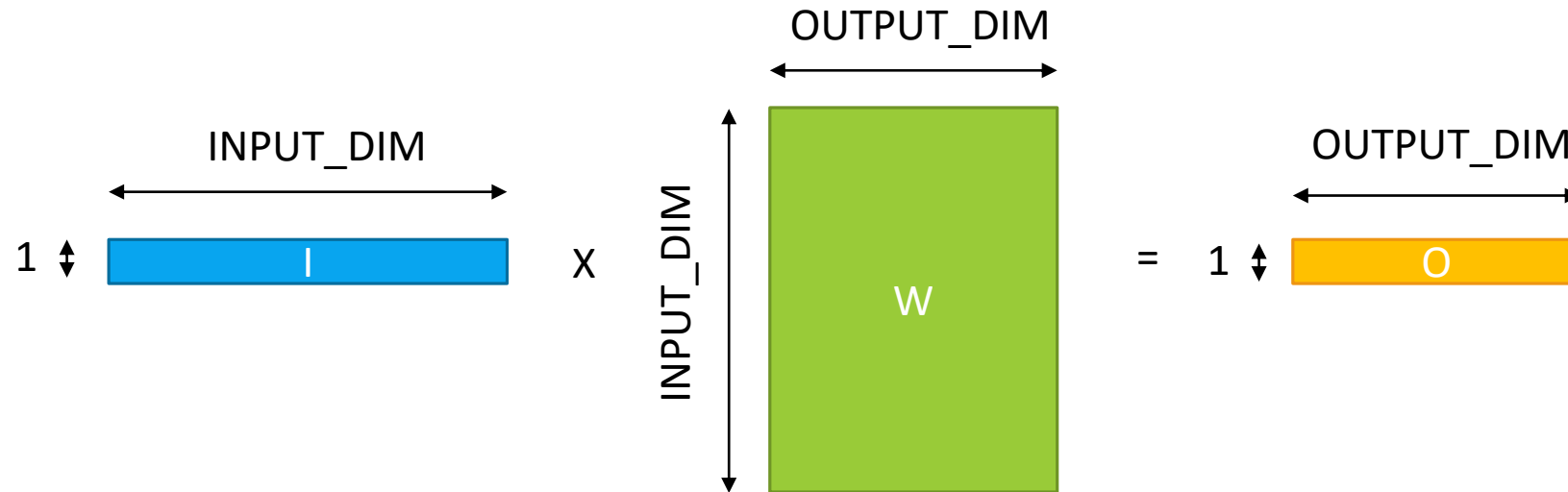
Weights Loading Architecture



Fully connected layers

Fully Connected Layers

- A fully connected layer takes one input vector and performs a matrix multiplication between the input vector and a weight matrix in order to produce another output vector.



$$O = I \times W$$

$$O_o = \sum_{i=0}^{INPUT_DIM} I_i W_{io}$$

Fully Connected Layers

Main concepts:

- The core of the module is a matrix multiplication between one input vector and a weight matrix
- Fully connected layers usually account for most of the weights of a CNN (even millions of weights!)
- For each input sample, every weight is used only once

Forms of parallelism:

- Each value of the output vector is independent from the others
- We can consider each output value as an accumulator that at each iteration takes the sum between the i-th input value and the corresponding weight

```
double input_vector[INPUT_DIM];
double weights[INPUT_DIM][OUTPUT_DIM];

double output_vector[OUTPUT_DIM];

for(int o=0; o < OUTPUT_DIM; o++)
{
    output_vector[o] = 0;

    for(int i=0; i < INPUT_DIM; i++)
        output_vector[o] += input_vector[i] * weights[i][o];
}
```

Fully Connected Layers with binary weights and activations

As already said, we can compute multiplications using the XNOR operations between two bits (0/1)

- Accumulating 1s and -1s is different than accumulating 1s and 0s. How can we make things right again?
- Every output value coming from the matrix multiplication is later normalized (batch normalization) and made binary again using a non linear activation (sign function). We can therefore implement these two steps by using a threshold, that accounts for the batch normalization parameters and the fact that the values are [0,1] and not [-1, 1]
- With values that are ± 1 , the range of each output accumulator is $[-\text{INPUT_DIM}, \text{INPUT_DIM}]$. With values that are 0 or 1, the range becomes $[0, \text{INPUT_DIM}]$ (only additions are performed -> simple counters on FPGA!)
 - If we don't consider the batch normalization factor, the new activation function has to become $\text{sign}(\text{output_accumulator} - \text{INPUT_DIM} / 2)$
- Fortunately, we can compute each threshold at training time. On the FPGA part we just have to compare each output accumulator with its corresponding threshold and output 1 if $\text{acc} \geq \text{threshold}$, 0 otherwise.

Fully Connected Layer on Vivado HLS

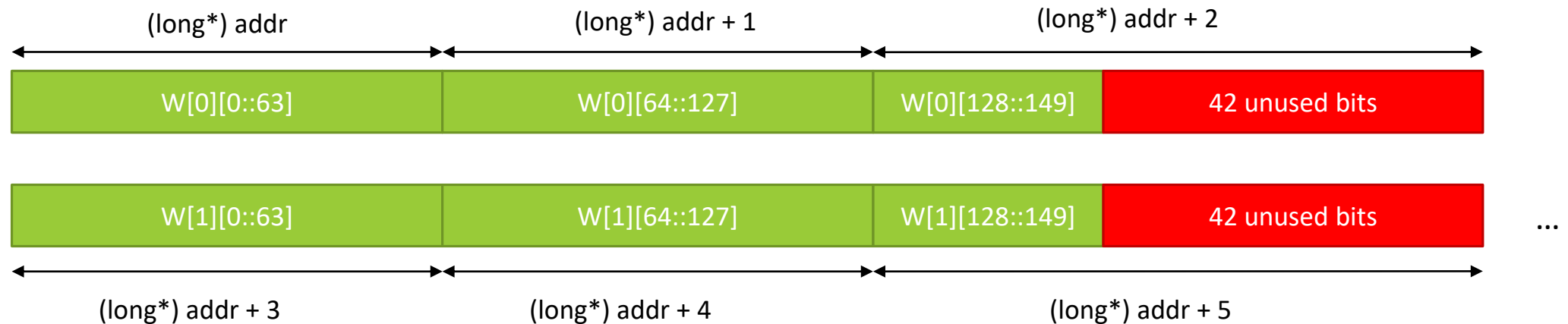
- Input and output AXI-STREAM of 1 bit
 - We can start doing computation as soon as we get just one value / we can write one output value as soon as we computed it
- Two ports are used for storing the weights:
 - 1 AXI (Lite) Slave interrupt, sent from a master to state that new weights are coming in
 - 1 64-bit AXI-STREAM input channel where the weights will be read
- Pseudocode:

```
void dense_layer (input_stream, output_stream, weights_interrupt, weights_addr)
{
    if (weights_interrupt)
    {
        weights_interrupt = 0;
        // LOAD WEIGHTS INTO ON-CHIP MEMORY FROM THE OFF-CHIP RAM
    }

    // PERFORM MATRIX MULTIPLICATION
}
```


Fully Connected Layer on Vivado HLS - Loading weights (1)

- First the boring part: storing weights!
- How do the weights need to be stored in the off-chip RAM?
 - Every row of the weight matrix needs to be stored in sequential cells of 64 bit
 - The number of 64 bit cells required by one row is $\text{OUTPUT_DIM} / 64$ if OUTPUT_DIM is divisible by 64, $\text{OUTPUT_DIM} / 64 + 1$ otherwise (to store the remaining most significant bits)
 - Example with $\text{OUTPUT_DIM} == 150$ ($64 + 64 + 22$):



Fully Connected Layer on Vivado HLS - Loading weights (2)

- Not only we have to load the matrix weights, but also the OUTPUT_DIM thresholds
- Each threshold is a unsigned value, with bit width $\lceil \log_2(INPUT_DIM + 1) \rceil$
- To make things easy, the thresholds are stored immediately after the last cell of 64 bits used for the weight matrix. Each cells contains only one threshold in its lower bits (our upper limit is therefore INPUT_DIM = $2^{64} - 1$: should be enough even for a omniscient neural net)

How all of this is handled on the FPGA part?

- The module reads the weights on the 64 bit AXI Stream port and updates them accordingly in the internal BRAMs
- This process could take some time (a couple of millisecond), but we are not in a hurry when loading weights

Fully Connected Layer on Vivado HLS - Matrix multiplication

Two implementations have been provided; their pros and cons will be analyzed soon.

First implementation:

- We can store OUTPUT_DIM accumulators (initialized at zero), and each time a new input bit comes in we are going to compute OUTPUT_DIM xnor_and_accumulate operations in parallel
 - Basically we have OUTPUT_DIM counters and at each clock the counter is increased only if the XNOR between the input bit and the corresponding weight is 1
- When the last input bit comes in, we can perform the thresholding operations (sequentially in order to save resources)
- Timing: 1 clock for accumulator resetting, INPUT_DIM clocks to perform the matrix multiplication, OUTPUT_DIM clocks for thresholding

Fully Connected Layer on Vivado HLS - Matrix multiplication

Having OUTPUT_DIM accumulators and performing OUTPUT_DIM xnor_and_accumulate in parallel takes a lot of resources and is intrinsically dependent on the value of OUTPUT_DIM. For large values, we might not have enough resources. This is why there's a second implementation.

Second implementation:

- Instead of having OUTPUT_DIM accumulators, we can keep only P_OUT (parallel out) accumulators ($\text{OUTPUT_DIM} \bmod \text{P_OUT} = 0$) and perform P_OUT xnor_and_accumulate operations in parallel at each clock.
- In order to do that we first have to read and store the whole input vector, otherwise we wouldn't be able to perform the computation of the next outputs. In fact, the next batch of P_OUT accumulators will use the same resources. The input vector is therefore cached when we compute the first batch of accumulators; the next batch will use the cached version of the input vector.
- Timing: $\text{OUTPUT_DIM} / \text{P_OUT}$ clocks for resetting the counters, $\text{OUTPUT_DIM} / \text{P_OUT} * \text{INPUT_DIM}$ clocks to perform the matrix multiplication, OUTPUT_DIM clocks for thresholding

Fully Connected Layer on Vivado HLS – Comparison of the two implementations

First implementation:

- Pros:
 - It's the fastest possible implementation (if we consider to receive the input bits sequentially)
- Cons:
 - We use the largest number of resources
 - Since it's dependent on the value of OUTPUT_DIM, for larger values it will be impossible to fit the module on the FPGA

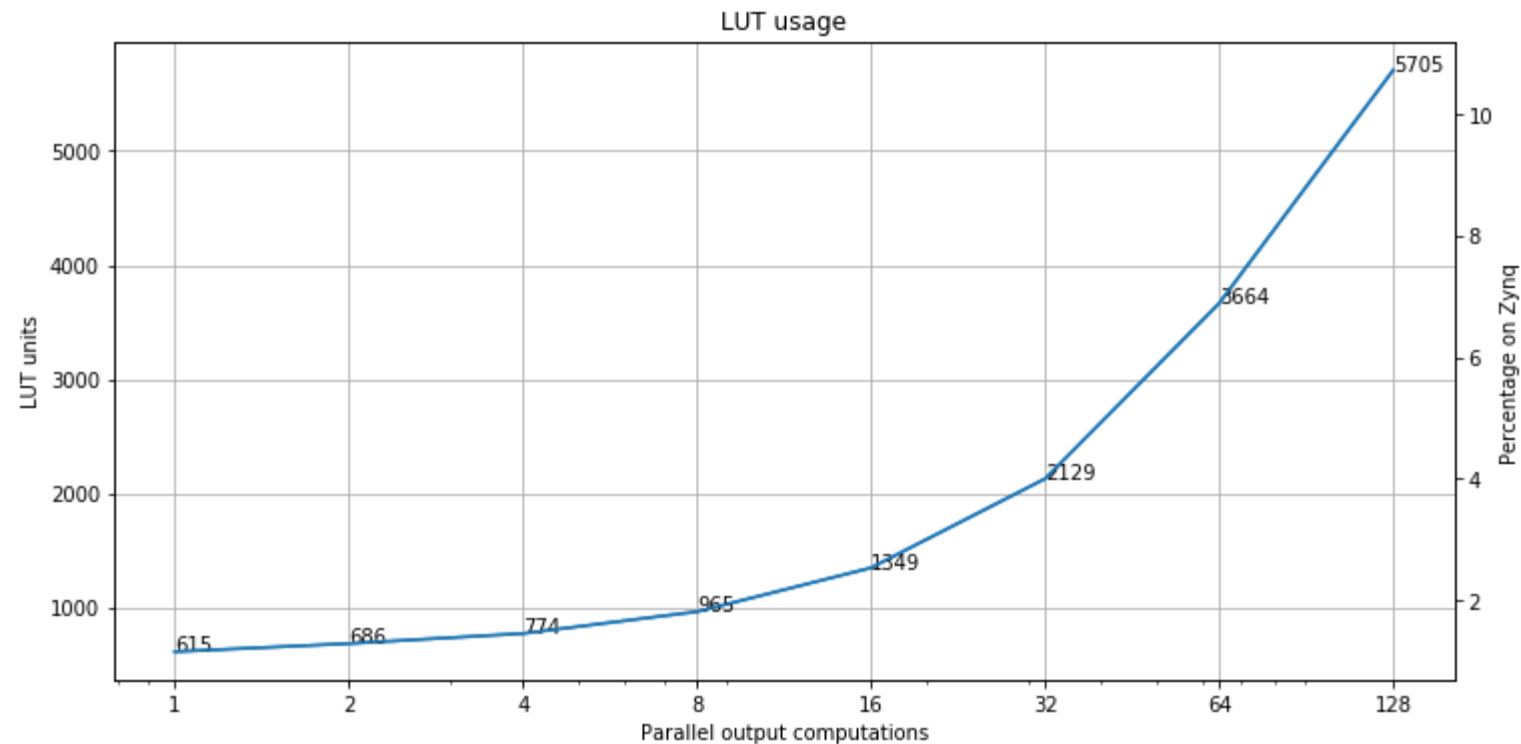
Second implementation:

- Pros:
 - The number of resources depends on P_OUT and not on OUTPUT_DIM (or on the matrix size, in general)
 - We can save resources by decreasing P_OUT (at the cost being slower)
- Cons:
 - Not the best solution if we want the best throughput and we don't need to care about resources

Fully Connected Layer on Vivado HLS – Resource and timing

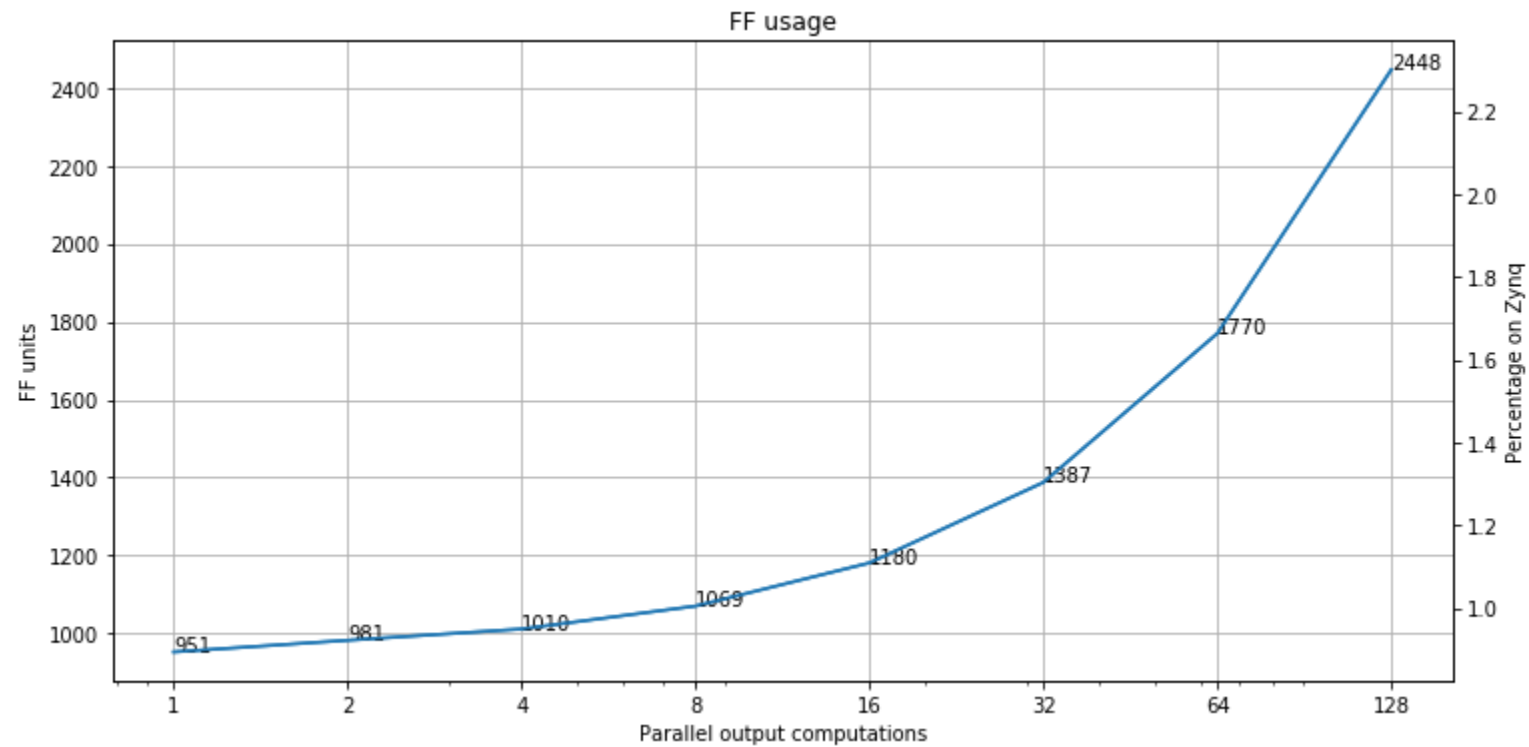
- In the following graphs we can see how the two implementations perform
- The results come from an implementation of the full module using INPUT_DIM = 3136 and OUTPUT_DIM = 128 (tot 400k weights). This is the first fully connected layer of our target CNN.
- When P_OUT (on the x axis) is equals to 128, the first implementation has been used; when P_OUT is less than 128, the second implementation has been used.
- The resources refer to the whole module (considering also the logic required for reading and storing the weights from the off-chip RAM)
- Target frequency is 100 MHz (10 ns) but the expected clock is 5.26 ns (almost 200 MHz)
- BRAM usage is equals to 32 out of 220 BRAMs for the second implementation, 31 BRAMs for the first impelmentation (since we don't have to cache the input vector)

Fully Connected Layer on Vivado HLS – Resource and timing

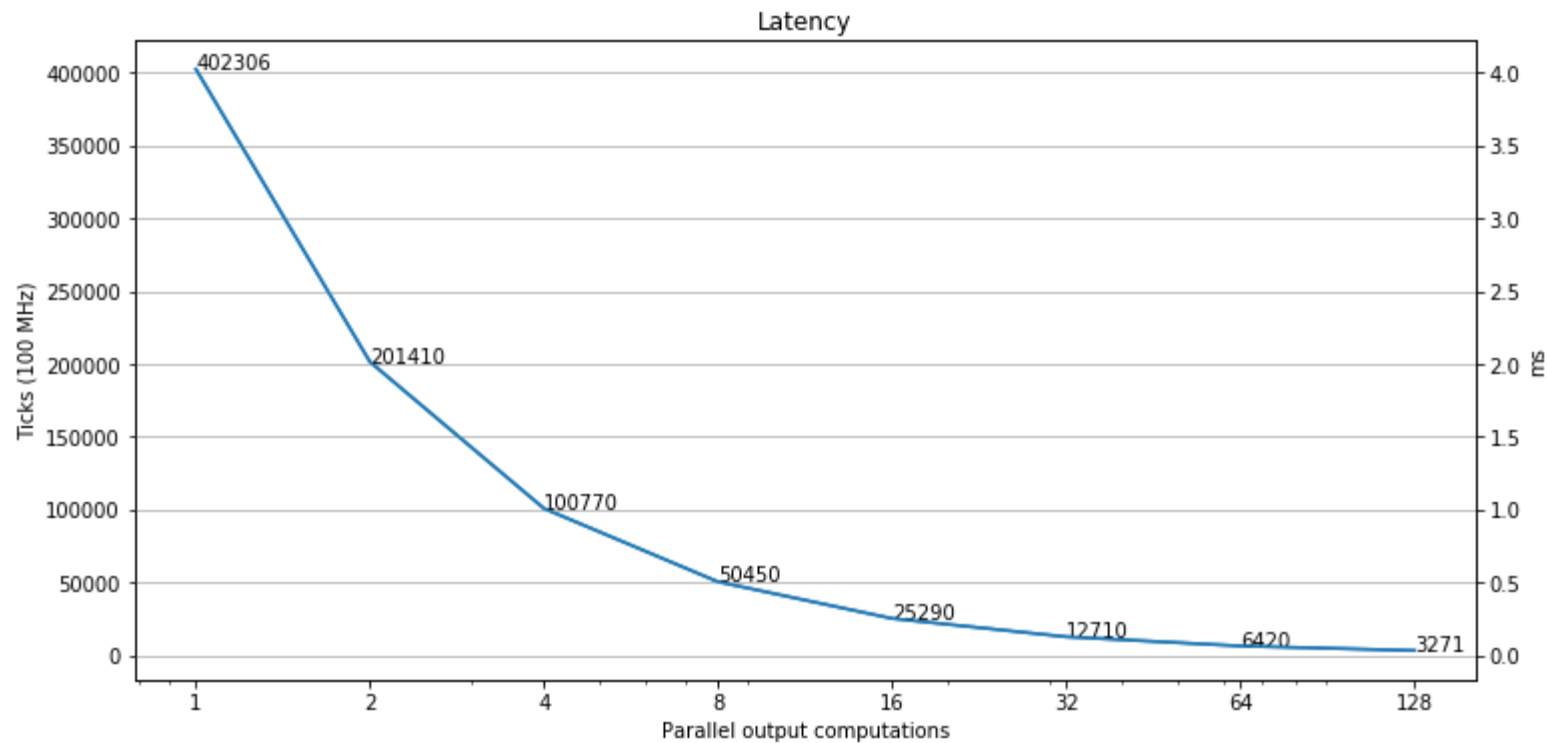


We can see that, as expected, the number of LUTs grows linearly with P_OUT (the x axis is in log scale)

Fully Connected Layer on Vivado HLS – Resource and timing



Fully Connected Layer on Vivado HLS – Resource and timing



Even with P_OUT = 32 we can perform a matrix multiplication of 400k weights in 0.13 ms!! (using only 4% of the available LUTs)

Fully Connected Layer on Vivado HLS – Resource and timing

In this second report, we show how the resources and timing change using a constant value of $P_OUT = 32$ (only the second implementation is shown):

- The number of LUTs is essentially independent from the size of the weight matrix.
- We can fit approx 4M weights before running out of BRAMs.

INPUT DIM	OUTPUT DIM	TOTAL WEIGHTS	FF	FF %	LUT	LUT %	BRAM	BRAM %	LATENCY (ms)
64	64	4k	740	0.7	1302	2.4	2	0.7	0.002
128	128	16k	869	0.8	1416	2.7	5	1.8	0.007
256	256	65k	1076	1.0	1541	2.9	9	3.2	0.024
512	512	262k	1411	1.3	1728	3.2	16	5.7	0.088
1024	1024	1M	2000	1.9	1875	3.5	59	21.1	0.341
1536	1536	2M	2545	2.4	1936	3.6	174	62.1	0.757
2048	2048	4M	3103	2.9	2057	3.9	231	82.5	1.337

Handling the classification layer

The last fully connected layer (a.k.a. the classification layer) is different from the other ones:

- Activation function is not the sign function, but the softmax
 - A typical approach is to compute the sigmoid on the CPU; this allows to save a lot of FPGA resources that would be used for floating point operations
- Batch normalization is still used in the paper, but without the sign output function we cannot use the thresholding approach

Different approaches to handle this situation:

1. Train a network without the last batch normalization step and see if it works: last operation before the softmax is therefore a matrix multiplication
2. Create another FPGA module that computes a real batch normalization (or embed the batch normalization operation inside this module)
3. Compute the last batch normalization on the CPU (the number of parameters depends only on the number of classes, so it shouldn't be computationally demanding)

What has been done for now:

- Added the possibility to synthesize a fully connected layer without the thresholding operation (matrix multiplication only); output values become the accumulators (width depends on INPUT_DIM)
- Training without the last batch normalization step may result in slower training and poorer accuracy (1-2% accuracy loss on MNIST)

Handling bigger networks

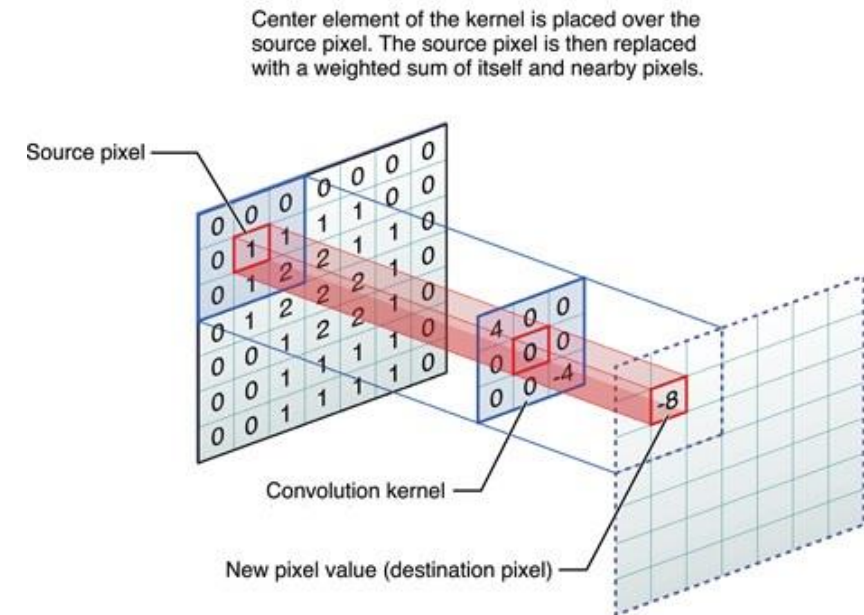
Bigger networks may have millions of weights in the last fully connected layers, and saving all the (even binary) weights on the FPGA resources may become infeasible

- The design could be modified by allowing to store only a subset of the weights on the FPGA resources
 - E.g. a couple of rows, one row at a time or even P_OUT weights at a time
- We need to cyclically load all the weights every time we seek to classify a new example -> delays due to the loading phase of the weights from DDR to PL
- With P_OUT and the amount of buffered weights we can basically control all parameters of our fully connected module (LUTs, BRAMS, timing)

Convolutional layers

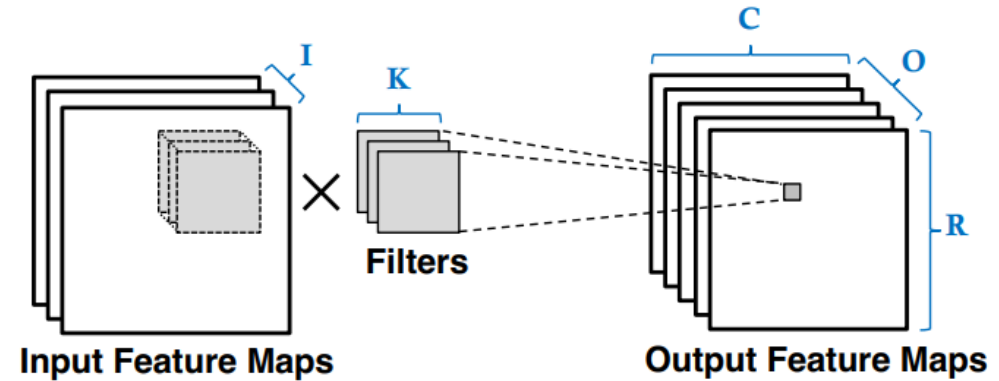
Image Convolution

- Given an input image \rightarrow 2D array of size (H, W)
- And a kernel \rightarrow 2D array of size (K, K)
- The 2D convolution between the image and the kernel returns a new image where each new pixel is the weighted sum of the kernel and the corresponding image window centered on the pixel itself
- We can think as the kernel sliding over the whole image and computing a new output pixel in each position.



Convolutional layers

- When talking about convolutional layers, we are not considering just one input image and one output but multiple input images and multiple output images



```
1 for (row=0; row<R; row++) {  
2   for (col=0; col<C; col++) {  
3     for (to=0; to<O; to++) {  
4       for (ti=0; ti<I; ti++) {  
5         for (ki=0; ki<K; ki++) {  
6           for (kj=0; kj<K; kj++) {  
             output_fm[to][row][col] +=  
               weights[to][ti][ki][kj]*input_fm[ti][S*row+ki][S*col+kj];  
           }  
         }  
       }  
     }  
   }  
}
```

Loop over output pixels
Loop over output feature maps
Loop over input feature maps
Loop over filter pixels

Convolutional layers

Here we are going to call F_{IN} (fan-in) the number of input feature maps and F_{OUT} (fan-out) the number of output feature maps

Some notes about the convolution weights (kernel):

- They are independent on the image size (H and W). In fact, the total number of weights is:

$$F_{OUT} \times F_{IN} \times K^2$$

- While in a dense (fully connected) layer each weight is used just once, in a convolutional layer we have to use all weights every time we move the kernel on the image -> that's a lot of computation to handle!

Convolutional layers with binary weights and activation on FPGA

- As with fully connected layers with binary weights and activations, we can replace every multiplication with a simple XNOR (equivalence) operation between two bits
- All the considerations that we discussed about the difference between having -1s and 1s vs 0s and 1s in fully connected layers apply here without a single change:
 - Every output map has different batch normalization parameters, which means that for every output map we have to define a different threshold
- There are multiple sources of parallelism. In fact, we could theoretically unroll all the loops:
- In a real implementation, we have to choose carefully which degrees of parallelism to pursue by considering the constraints on our HW resources

```
1 for (row=0; row<R; row++) {  
2   for (col=0; col<C; col++) {  
3     for (to=0; to<O; to++) {  
4       for (ti=0; ti<I; ti++) {  
5         for (ki=0; ki<K; ki++) {  
6           for (kj=0; kj<K; kj++) {  
             output_fm[to][row][col] +=  
               weights[to][ti][ki][kj]*input_fm[ti][S*row+ki][S*col+kj];  
           }  
         }  
       }  
     }  
   }  
}
```

Loop over output pixels
Loop over output feature maps
Loop over input feature maps
Loop over filter pixels

Convolutional layers in Vivado HLS

In our implementation we expect to receive input pixels in a sequential manner:

- Row and col loops are therefore not unrolled
- The resources are (mostly) independent on the image size
- The total latency is proportional to the image size (the minimum latency is the size of the image itself)

I/O of the convolution module:

- Input is a AXI-STREAM with F_{IN} bits
- Output is a AXI-STREAM with F_{OUT} bits

As for fully connected layers, there are two ports for handling the weights loading:

- 1 AXI (Lite) Slave interrupt, sent from a master to state that new weights are coming in
- 1 64-bit AXI-STREAM input channel where the weights will be read

Convolutional layers in Vivado HLS – Loading weights

Inside the convolution module, weights belonging to the same KxK filter are flattened.

- Weights type in HLS becomes `ap_uint<K*K>`
 - This allows to avoid writing multiple nested loops and to state explicitly that all weights belonging to the same KxK filter need to be accessed in parallel. The KxK vectors will be therefore packed accordingly inside the FPGA resources
1. When loading weights, the module expects to receive all the binary weights of the same KxK filter in one 64 bit input value:
 - This implementation choice limits K to a maximum value of 8 (for smaller values the most significant bits are discarded)
 - Typical values are smaller (3, 5), but this design choice could be modified by imposing multiple reads
 2. After $F_{OUT} \times F_{IN}$ read operations, we need other F_{OUT} operations in order to read the thresholds
- $F_{OUT} \times (F_{IN} + 1)$ read operations in total.

Convolutional layer in Vivado HLS

- The HW structure behind the core of the computation is the same for a usual convolution (line buffers, sliding window)
 - This is where the width of the input feature maps can influence the number of resources (the higher the width, the longer the line buffers)
- Every time a new set of F_{IN} input 'pixels' comes in:
 1. The sliding window and the line buffers are updated
 2. The F_{OUT} binary values at the current location are computed by performing $F_{OUT} \times F_{IN}$ *xnor_and_popcount* and F_{OUT} thresholding operations
 3. The result is written on the output stream

Convolutional layers in Vivado HLS: *xnor_and_popcount*

The core of the convolution operations is called *xnor_and_popcount*:

- Given one input window of $K \times K$ bits relative to one input map
- And one weight filter of $K \times K$ bits
- The *xnor_and_popcount* performs a bitwise xnor (equivalence) between corresponding input and weight bits and counts the number of ones
- The result is a unsigned number ranging from 0 to $K \times K$
 - Results for the same output map will be accumulated in a single accumulator with bit width $\lceil \log_2(F_{IN} \times K^2 + 1) \rceil$. This accumulator will then be thresholded in order to compute the output bit
- Each time we move the input window in a new position, we have to repeat this basic operation $F_{OUT} \times F_{IN}$ times
- With increasing values of F_{OUT} and F_{IN} (they could reach even 256, 512 or 1024) it is unfeasible to perform all *xnor_and_popcount* and accumulate operations in just 1 clock
- We need to be able to define the degree of parallelism that we seek to obtain (how many *xnor_and_popcount* per clock) and control the resulting number of resources

Convolutional layers in Vivado HLS

- We define two parameters, P_{OUT} and P_{IN} (parallel out and parallel in) that define how many operations we want to perform in 1 clock
 - E.g. $F_{IN} = F_{OUT} = 32$ -> we need to perform, in each image position, $32*32 = 1024$ *xnor_and_popcount* and accumulate the partial results in 32 output accumulators, before thresholding them
 - By specifying , $P_{OUT} = 4$ and $P_{IN} = 8$ we say that we want to deal with blocks of 4 output values. We are going to perform a total of $4*8=32$ *xnor_and_popcount* at a time (using 8 input maps per output map), accumulate the results in 4 accumulators
 - The time to compute , P_{OUT} output values is F_{IN} / P_{IN}
 - The time to compute F_{OUT} output values becomes $\frac{F_{OUT} \times F_{IN}}{P_{OUT} \times P_{IN}}$
- With these values we are able to control the resources used
 - Resources grow with , $P_{OUT} \times P_{IN}$
 - Timing is inversely proportional to $P_{OUT} \times P_{IN}$. The total (approx) latency of the module for an entire image becomes $H \times W \times \frac{F_{OUT} \times F_{IN}}{P_{OUT} \times P_{IN}}$
 - More resources <--> faster computation!

Convolution structure in HLS – First draft (pseudocode)

```
for(int row=0; row < IMG_HEIGHT; row++)
  for(int col=0; col < IMG_WIDTH; col++)
  {

    // Read input values, shift line buffers and sliding window

    for(p_out_it=0; p_out_it < FAN_OUT / P_OUT; p_out_it++)
    {

      for(p_in_it=0; p_in_it < FAN_IN / P_IN; p_in_it++)
      {
        // Perform P_OUT * P_IN xnor_and_popcounts and accumulate values
      }

      // Threshold the P_OUT accumulators and compute P_OUT output bits

    }

    // All output bits for the current position have been computed; write the result on the output stream
  }
```

Convolutional layer in Vivado HLS: pipelining

The objective is to stay inside the column loop for $\frac{F_{OUT} \times F_{IN}}{P_{OUT} \times P_{IN}}$ clocks

- Pipelining would solve the problem!

However, we cannot achieve the desired result wherever we apply the PIPELINE directive:

1. If we apply the pragma inside the inner most loop (p_in_it), the other parts of the architecture (line buffer updating, thresholding) are not pipelined
2. If we apply the pragma to the column loop, all inner loops become fully unrolled, even the p_out_it and p_in_it loops!

We need to write the code differently, making sure that the whole structure gets pipelined and the convolution loops are not unrolled

- We can achieve that by moving the whole logic inside the inner most loop and conditioning it using the values of p_out_it and p_in_it

Convolution structure in HLS pipelined (pseudocode)

```
for(int row=0; row < IMG_HEIGHT; row++)
  for(int col=0; col < IMG_WIDTH; col++)
    for(p_out_it=0; p_out_it < FMAPS_OUT / P_OUT; p_out_it++)
      for(p_in_it=0; p_in_it < FMAPS_IN / P_IN; p_in_it++)
      {
        #pragma HLS PIPELINE


        if (p_out_it == 0 && p_in_it == 0)
        {
          // Read input values, shift line buffers and sliding window
        }

        // Perform P_OUT * P_IN xnor_and_popcounts and accumulate values

        if (p_in_it == FMAPS_IN / P_IN - 1) // End of the inner loop
        {
          // Threshold the P_OUT accumulators and compute P_OUT output bits
        }

        if (p_out_it == FMAPS_OUT / P_OUT - 1 && p_in_it == FMAPS_IN / P_IN - 1) // End of the outer loop
        {
          // All output bits for the current position have been computed; write the result on the output stream
        }
      }
}
```

Convolution structure in HLS

- With this code we are able to control the output latency of the convolution module
- Unfortunately, we are not able to achieve II=1 with a clock of 10 ns for the following reasons
 - A read-write dependency is found on the line buffer; can be overridden using the DEPENDENCY pragma but needs further analysis to see if the behavior is not altered
 - Some worst-case paths require a clock period of at least 13 ns
- At the time, two solutions are possible
 - Achieve II=2 with clock of 10 ns. The total latency becomes $2 \times H \times W \times \frac{F_{OUT} \times F_{IN}}{P_{OUT} \times P_{IN}}$
 - Achieve II=1 with a slower clock, using the DEPENDENCY pragma
- Further work could solve this problem with some HLS tricks... 

Required number of BRAMs to store convolution weights

- Each BRAM can hold up to 18kbit. It can be configured to have a predefined size and bit width (e.g. 2K elements of 9 bits, 1K elements of 18 bits, ...). The maximum configurable parallelism is 512 elements of 36 bits
- Given the convolution layer parameters F_{IN} , F_{OUT} , K , P_{IN} , P_{OUT} we want to find the minimum number of BRAMs required to synthesize the required architecture
 1. $K = 3$ is supposed here
 2. We need to access in one clock $P_{IN} \times P_{OUT}$ values of 9 (K^2) bits/weights; therefore the minimum number of 36-bit wide BRAMs to obtain the following parallelism is $Z = P_{IN} \times P_{OUT} \times 9 \div 36 = \frac{1}{4} P_{IN} \times P_{OUT}$
 - This is the minimum required number of BRAMs to achieve a desired parallelism; it does not depend on the size of the convolution!
 3. To save all weights we need at least $\left\lceil \frac{F_{OUT} \times F_{IN} \times 9}{18k} \right\rceil = \left\lceil \frac{F_{OUT} \times F_{IN}}{2048} \right\rceil$ BRAMs
 4. The overall required number of BRAMs becomes:

$$\max \left(\left\lceil \frac{F_{OUT} \times F_{IN}}{2048} \right\rceil, \left\lceil \frac{1}{4} P_{IN} \times P_{OUT} \right\rceil \right)$$

Required number of BRAMs to store convolution weights

		Convolution parallelism ($P_{IN} \times P_{OUT}$)					
Size of the convolution $F_{OUT} \times F_{IN}$		4	8	16	32	64	128
	8x8	1	2	4	8	16	32
	16x16	1	2	4	8	16	32
	32x32	1	2	4	8	16	32
	64x64	2	2	4	8	16	32
	128x128	8	8	8	8	16	32
	256x256	32	32	32	32	32	32

- The table shows the number of BRAMs required given the convolution size and the parallelism that we want to achieve (again we consider K=3)

Weights definition in HLS

```
static ap_uint<K2> weights[FAN_OUT/ P_OUT][FAN_IN/ P_IN][P_OUT][P_IN];
```

```
#pragma HLS ARRAY_RESHAPE variable=weights block factor=4 dim=4  
#pragma HLS ARRAY_PARTITION variable=weights complete dim=3  
#pragma HLS ARRAY_PARTITION variable=weights complete dim=4
```

- The more natural definition of the weights array would be `ap_uint<K2> weights[FAN_OUT][FAN_IN]`. In our definition as a 4D array we can explicitly control the degree of parallelism (P_{IN} and P_{OUT})
- The RESHAPE pragma is used to pack adjacent 9 bit ($K \times K$) values into a single 36 bit value, allowing to use the full BRAM width
 - This pragma is optimized for $K=3$!!!
 - Without this pragma HLS uses a lot more BRAMs than necessary (2x or even 4x !!)
- The PARTITION pragmas state that we want to access in parallel blocks of $P_{OUT} \times P_{IN}$ vectors
- HLS raises a warning saying that reshaping and partitioning the same array may lead to unexpected synthesis behavior, but it seems to work anyways...

Conv Layers - Resource Usage

- $F_{IN} = F_{OUT} = 128$ (image size 32x32)
 - 8 BRAMs are needed for line buffers
- Given the required parallelism ($P_{IN} \times P_{OUT}$), resources seem to grow when P_{OUT} grows
- To minimize resources, maximize P_{IN} is the easy option

		P_{OUT}			
		1	2	4	8
P_{IN}	4	16 BRAM 3788 FF 2393 LUT	16 BRAM 4035 FF 2916 LUT	16 BRAM 4480 FF 3654 LUT	16 BRAM 4931 FF 5139 LUT
	8	16 BRAM 3865 FF 2641 LUT	16 BRAM 4206 FF 3437 LUT	16 BRAM 4794 FF 4602 LUT	24 BRAM 5589 FF 7067 LUT
	16	16 BRAM 4002 FF 2729 LUT	16 BRAM 4473 FF 4302 LUT	24 BRAM 5404 FF 6425 LUT	40 BRAM 6821 FF 10755 LUT
	32	16 BRAM 4295 FF 3695 LUT	24 BRAM 5057 FF 6176 LUT	40 BRAM 6567 FF 10223 LUT	
	64	24 BRAM 5759 FF 5622 LUT	40 BRAM 6648 FF 9894 LUT		

Conv layers – Optimized version

- To save resources, an «optimized» version has been developed; this version assumes $P_{IN} = F_{IN}$ and $P_{OUT} = 1$
- This was also done since the setting $P_{IN} = F_{IN}$ in the normal implementation was breaking the synthesis
- This implementation allows us to save even more resources and decrease the clock period to 10.9 ns
 - It's still above our target value (10ns), but it is to say that Vivado HLS predictions are usually overestimated and a real synthesis into Vivado should make this problem disappear.

Conv Layers – A note about padding

- The implemented module doesn't apply any padding (padding='valid' in TensorFlow)
- To apply padding, see the next slides

Fixed Point Convolution

The first convolutional layer of the BinaryNet is an exception to the «fully binarized» idea:

- Its input feature maps (the input image) are not binary, while weights and output feature maps are still binary
- The core part of the convolution becomes a multiplication between a non-binary value and another one with value -1 or 1: this can be expressed as a simple sign inversion!
 - After that, values need to be accumulated as usual
- A good trade-off between binary computation and floating-point computation is using fixed-point values
 - In fixed-point values, the bit width of the value is split into parts: one for the integer part and one for the decimal part
 - The number of bits used for the integer and decimal parts are known at compile-time and never change
 - This allows to synthesize resource-efficient operations in FPGAs

Fixed Point Convolution

- Typical values of F_{IN} for the first convolutional layer are 1 or 3 (grayscale or RGB image)
- This means that the total computation required $F_{IN} \times F_{OUT}$ is much lower than the one required in the following fully binary layers
- With much less computation required, we can be fast (compared to the next layers) without exploiting much parallelism and therefore resources

The implementation of the module is a simplification of the binary convolution:

- In this implementation $F_{IN} = P_{IN}$ are hardcoded to 1. This means that we only need to care about P_{OUT}
- Weights and thresholds become simple 1D arrays, with size equals to F_{OUT}
- To save resources, we can set a cyclic ARRAY PARTITION directive to both arrays with factor equals to P_{OUT} .

Example with $P_{OUT} = 8$:

THRESH/WEIGHTS	complete	cyclic factor=8
complete	1541 FF, 3413 LUT	1507 FF, 3269 LUT
cyclic factor=8	1519 FF, 3197 LUT	1507 FF, 3036 LUT

Fixed Point Convolution

- All fixed point values (inputs, accumulators and thresholds) have a total bit width of 13, with 5 bits for the integer part
 - Input values are originally floating points values between 0 and 1 (even 1 bit for the integer part would be enough for them)
 - With $F_{IN} = 1$, every accumulator needs to accumulate K^2 input values, that could be added or subtracted; the range of each accumulator becomes $[-K^2, +K^2]$, and 5 bits (including one sign bit) are necessary for $K = 3$
 - Since original images have discrete integer values between 0 and 255, using 8 bits for the decimal part should be enough; total size is $8+5 = 13$.
- Results for $F_{OUT} = 32$ and image size 32×32 are reported here. No BRAMs or DSPs are used, but resources grow much faster than fully binary layers with increasing values of P_{OUT} .
 $P_{OUT} = 4$ seems to be a good trade-off between latency and resource consumption

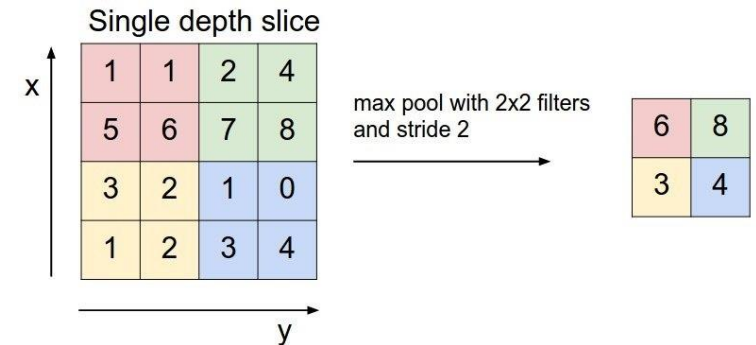
P_OUT	FF	FF%	LUT	LUT%
1	751	0.7%	795	1.5%
2	850	0.8%	1088	2.0%
4	1020	1.0%	1684	3.2%
8	1507	1.4%	3036	5.7%
16	1879	1.8%	5559	10.4%
32	3132	3.0%	9712	18.2%

Max Pooling

Max Pooling

Max Pooling is a very simple operation: given a 2D image and a window of size K , max pooling returns the maximum element of the image inside the window for all the positions occupied by the window on the image

- It's somehow similar to a convolution, but it doesn't require any weights
- Usually the stride (by how many pixels the window is moved every time) is equals to K , so we end up partitioning the image in $H/K * W/K$ non-overlapping regions. The output size of the image will be, in fact, H/K by W/K .
- When applied to a batch of feature maps (the output of a convolutional layer), max pooling is applied independently on all the feature maps. The output tensor size of the max pooling operation will therefore be $(H/K, W/K, \text{FMAPS})$

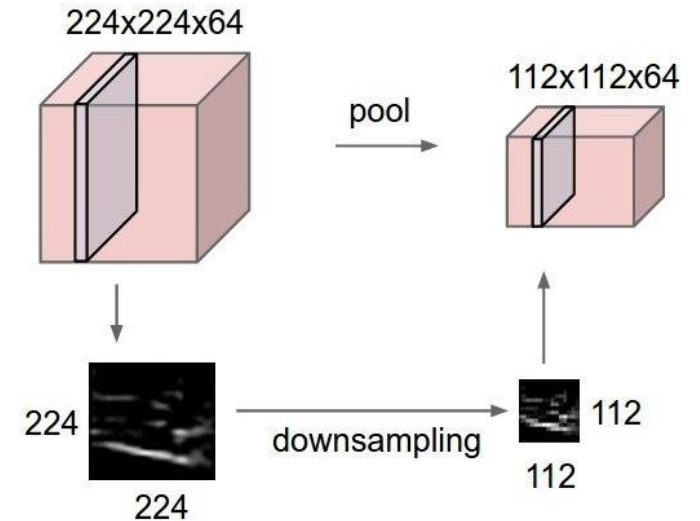


Max Pooling

Why do we need max pooling?

1. Reduces the input dimensionality of the following layers -> less computation
2. Allows to learn higher level features (objects, bigger entities inside the image) in the next layers

Since we are dealing with binary inputs, performing a max operation is equivalent to computing the OR over all the values inside the window along the first two dimensions:
(K, K, FMAPS) -> (FMAPS)



Max Pooling in Vivado HLS

- Input and output are both AXI Stream of bit width == FMAPS (in 1 clock we receive all the input maps in a certain position)
- Implementation of the Max Pooling operation on FPGA is quite straightforward: the underlying structure is the same used for the convolution operation (line buffers, moving window); we just need to change the operation that needs to be computed.
- Moreover, we need to compute the max only on particular positions inside the image. This is achieved by imposing the condition:

$$\text{row \% K} == \text{K} - 1 \ \&\& \ \text{col \% K} == \text{K} - 1$$

where row and col are the indexes of the last input value.

Max Pooling in Vivado HLS

Timing

- The module is able to compute the bitwise OR in one clock using pipelining, so the number of clocks required to perform Max Pooling is equals to the size of the input feature maps ($H*W$).

Resources

- A few resources need to be used to perform the bitwise ORs
- The number of resources grows with K and FMAPS. For FMAPS = $H = W = 32$ and $K = 2$ we just need 0.5% FF and 0.9% LUT (no BRAMs) on the Zynq.
- For larger values of W and K we will need to store more data on the line buffers, so BRAMs will be used too.

Additional considerations

- If H or W are not divisible by K, the last rows or columns will be discarded. This is also the default on TensorFlow (padding = 'valid' means no padding). To achieve padding (padding = 'same' on TF), we can add a padding module before the Max Pooling module without changing the line buffer logic inside this component.

2D Padding

2D Padding for Convolution

- So far we have considered convolution operations that apply no padding (padding='vaild' on TF) and therefore reduce the dimensionality of the otuput feature maps.
- If this is not the desired behavior, we can achieve padding by creating a Padding module and inserting it before a convolution module.
- Input and output is AXI Stream with bit width equals to FMAPS.
- Padding is applied on all four borders of the input images. This is the pseudocode:


```
for (int row = 0; row < H + 2*PAD; row++)
    for (int col = 0; col < W + 2*PAD; col++)
    {
        if (row >= PAD && row < H + PAD && col >= PAD && col < W + PAD)
            output = input
        else
            output = PADDING_VALUE
    }
```

- Since no caching or arithmetic is needed (except for the 4 comparisons), this module basically comes for free

2D Padding for Convolution

- With a convolutional layer of kernel size K , we want to set $PAD = (K - 1) / 2$ (K is supposed odd)
- We also have to change the input height and width inside the convolutional layer, that become $H + 2*PAD$ and $W + 2*PAD$
- The convolutional layer doesn't apply padding! But he's not aware that we are pretending that the image is bigger and we are feeding him some extra values..
- Why we didn't create a convolutional module that applies padding?
 - Because doing a separate module is easier!
 - Because adding the padding logic on the basic convolution structures (line buffer, moving window) is more complicated and it would probably require to slow down the whole process.

2D Padding for Convolution

- The padding values can be all 0s or all 1s. When training the network, this means that we have to apply padding of all -1s or all 1s.
- Traditionally all deep learning frameworks (TensorFlow included) allow two plug&play forms of padding: no-padding (padding='valid') or zero padding (padding='same')
 - Zero padding injects a third value (zero) into our binary net! 
- The training process needs to be changed by using different padding values (1s or -1s instead of 0s); the PADDING_VALUE has to be set with the same padding value (0 for -1s or 1 for 1s)
- In TensorFlow, we can achieve the desired behavior by using a specific method *pad* before the convolution operation:

```
# kh is half the kernel size
inputs = tf.pad(inputs, [[0,0], [kh, kh], [kh, kh], [0,0]], constant_values=1.)
```

The padding option in the convolution function has to be set to 'valid' (no padding); we are basically separating padding and convolution just like we did in our FPGA implementation.

- Experiments show that using 1s padding instead of zero padding doesn't affect the training quality!

Other modules

Sending results to the CPU

- After the last fully connected layer, a module named *classification_sender* is responsible of making the final values available to the CPU by saving them inside the off-chip DDR
- When all the values have been received, an interrupt signal is automatically issued to the CPU
- The CPU will register an interrupt handler that will read the values and compute the softmax
- Appropriate C drivers will be automatically generated to enable the interrupt

```
void classification_sender(hls::stream<in_type> &input_stream, uint* class_data)
{
    /*
     * With the pragma HLS INTERFACE s_axilite port=return
     * Vivado HLS infers an interrupt output port that gets high
     * when the function processed everything.
     * Also, the C functions for interrupt handling are inferred
     * inside the C drivers (SDK)
     */
    #pragma HLS INTERFACE s_axilite port=return bundle=BUS
    #pragma HLS INTERFACE m_axi depth=10 port=class_data bundle=BUS
    #pragma HLS INTERFACE axis port=input_stream

    static u32 buffer[NUM_CLASSES];

    for(int i=0; i < NUM_CLASSES; i++)
    {
        #pragma HLS PIPELINE II=1
        in_type in_val = input_stream.read();
        buffer[i] = in_val;
    }

    memcpy((uint*) class_data, (uint*) buffer, NUM_CLASSES * sizeof(uint));
}
```

TO-DO

- Work on the binary convolution to achieve $II=1$
- Choose how to implement the last fully connected layer (batch-norm or not? In FPGA or CPU?)
- Implement the WEIGHT_MUX module, responsible of reading weights from the DDR and sending them to the required module via AXI-Stream
- Implement a module that reads the input image in DDR (AXI-Master) and streams it to the first conv layer (AXI-Stream)
- Test each module separately with HLS co-simulation
- Connect everything together on Vivado
- Test with Vivado + SDK
 - How do we load weights into the DDR? (Hard-coded, File, Ethernet)

Resource and Timing estimation

Resources of implemented modules for our target CNN

- In the next slide, we report the timing and the number of used resources for each layer of our trained 460k-weights CNN for MNIST classification
- All modules have a target clock of 10 ns. Unfortunately, setting a pipeline directive with $II=1$ makes the design of the convolutional layer too slow according to HLS. For those, we set the pipeline directive to $II=2$ (2 times slower)
 - It is likely that a real implementation (on Vivado) will make this warning disappear

Resources of implemented modules for our target CNN

	Fixed Point Convolution	Padding	Binary Convolution	Max Pooling	Padding	Binary Convolution	Padding	Binary Convolution	Max Pooling	Fully Connected	Fully Connected	TOTAL
PARAMS	Image size 29x29 K = 3 FAN_IN = 1 FAN_OUT = 32 P_OUT = 4	Image size 28x28 PAD=1 FMAPS=32	Image size 29x29 K = 3 FAN_IN = 32 FAN_OUT = 32 P_IN = 32 P_OUT = 1	Image size 28x28 K=2 FMAPS=32	Image size 14x14 PAD=1 FMAPS=32	Image size 15x15, 14x14 K = 3 FAN_IN = 32 FAN_OUT = 64 P_IN = 32 P_OUT = 1	Image size 14x14 PAD=1 FMAPS=32	Image size 15x15, 14x14 K = 3 FAN_IN = 64 FAN_OUT = 64 P_IN = 64 P_OUT = 1	Image size 14x14 K=2 FMAPS=64	INPUT_DIM 3136 OUTPUT_DIM 128 P_OUT 32	INPUT_DIM 128 OUTPUT_DIM 10 P_OUT 10	
BRAM			8			8		16		32	1	97 (44%)
FF	1012	187	1529	303	183	1610	311	3575	553	1019	468	11 769
FF%	1.0%	0.2%	1.4%	0.3%	0.2%	1.5%	0.3%	3.4%	0.5%	1.0%	0.4%	11.1%
LUT	1664	206	2968	286	201	2989	297	4925	467	1865	523	16 525
LUT%	3.1%	0.4%	5.6%	0.5%	0.4%	5.6%	0.6%	9.2%	0.9%	3.5%	1.0%	31.1%
LATENCY (ms)	0.134	0.0002	0.538	0.0008	0.0005	0.288	0.0005	0.288	0.0002	0.127	0.0001	1.37

Resources and timing of the implemented CNN

- Considering our implemented modules, we achieve a total latency of 1.37 ms without even using half of the available resources!
 - Also, Vivado HLS resource predictions are usually over-estimated and a real FPGA implementation on Vivado could allow us to save even 5% of the resources
- We didn't consider that the whole latency would be higher because we also need to:
 - Read the image from the DDR to the FPGA
 - Write the results in DDR
 - Compute softmax into the interrupt handler of the CPU
- However, we have to consider that the whole design is pipelined!
 - Layer $K-1$ can process image $I+1$ while layer K processes image I .
- The throughput of the entire architecture depends on the slowest component of the pipeline
 - In our case, the slowest component is the first binary convolution, with latency=0.538ms
 - Thanks to pipelining, we can process $1000/0.538 \approx \mathbf{1850 \text{ images/sec}}$
- We could also choose to use increase the parallelism on the slowest module, thus making the whole pipeline faster
 - By doubling the resources on the first binary convolution ($P_OUT = 2$) and relying on the fact that a real implementation will allow us to achieve $II=1$ on the convolution module, we could process about 7000 images/sec !

Summary

- We were able to train a convolutional neural network for image classification with binary weights and activations on the MNIST dataset
- We were able to build the core layers (convolution, fully connected, max pooling) for a FPGA device using High-Level Synthesis Tools (Vivado HLS) and evaluate trade-offs between timing and resources
- Our network uses less than half the resources of a low-end Zynq device and can reach a throughput of thousands of images per second
- Python and HLS code is available here: <https://github.com/riccardoalbertazzi/bnn>