Hardware Acceleration of CNN using Reconfigurable logic:

-Darshan C G



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. [1] tried to reduce value types to the simplest and most efficient possible value: a binary value

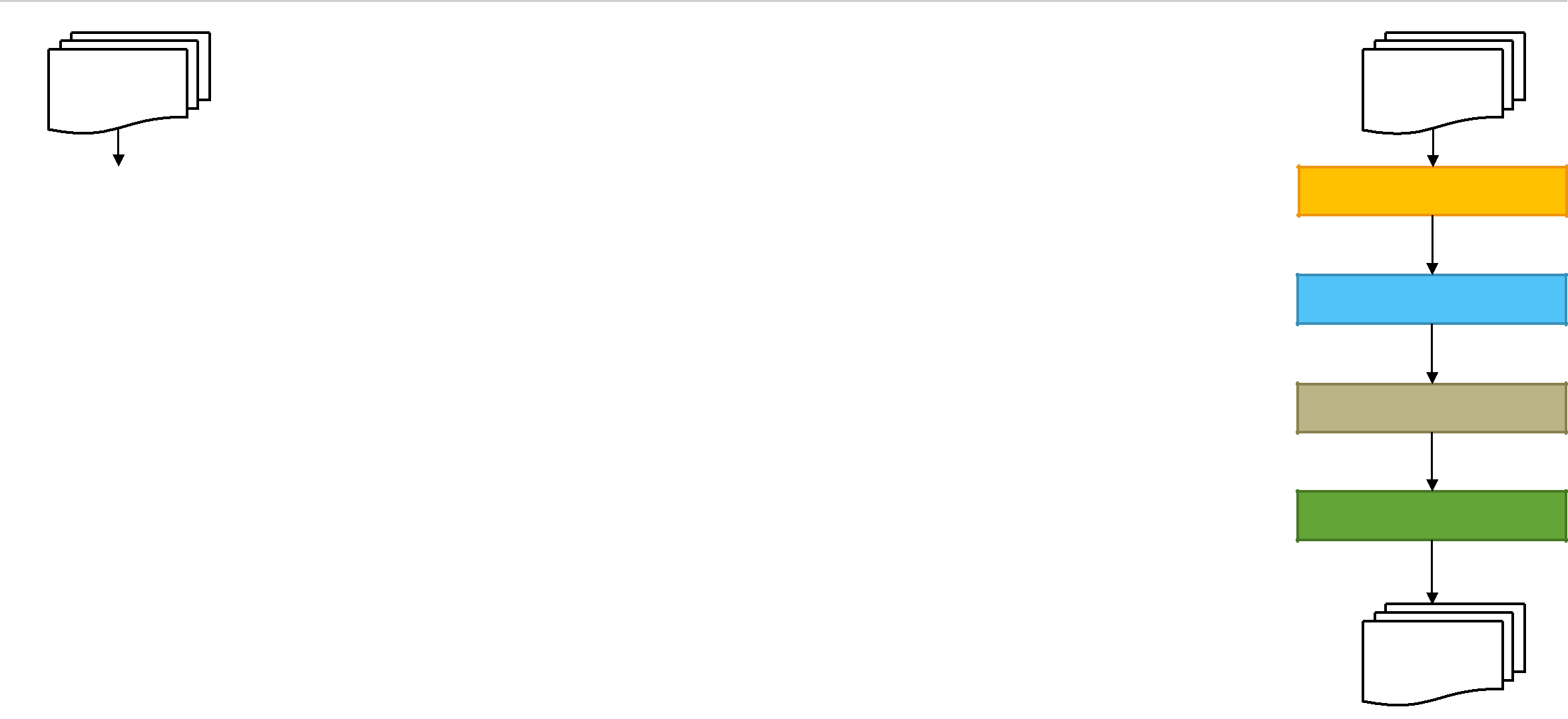
* In their implementation they propose a fully binarized convolutional network, named BinaryNet, where both weights and activations (inputs and outputs of each layer) have only two values: -1 or 1

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 [2]) 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

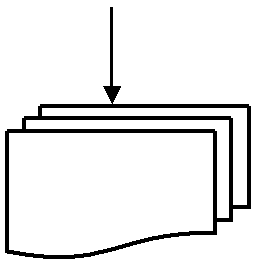
****

|  |
| --- |
| Paper structure |

M binary

maps

|  |  |  |
| --- | --- | --- |
| CONVOLUTION | • In the original paper the basic convolution structure consists |  |
|  | of the following ordered layers: Convolution, Max Pooling, |  |
|  | Batch Normalization, Activation Function (sign function) |  |
| MAX POOLING | • As suggested in [3], the Max Pooling operation has been |  |
|  | moved after the activation function (and before the next |  |
|  | convolution) |  |
| BATCH NORM |  |
| • By doing that, we let the Max Pooling operation work |  |
|  | with binary values instead of integer values -> less |  |
|  | resources (see later) |  |
| BINARIZATION (SIGN) |  |
|  |  |
|  |  |  |



1. binary maps

CONVOLUTION

BATCH NORM

BINARIZATION (SIGN)

MAX POOLING

|  |
| --- |
| Project structure |

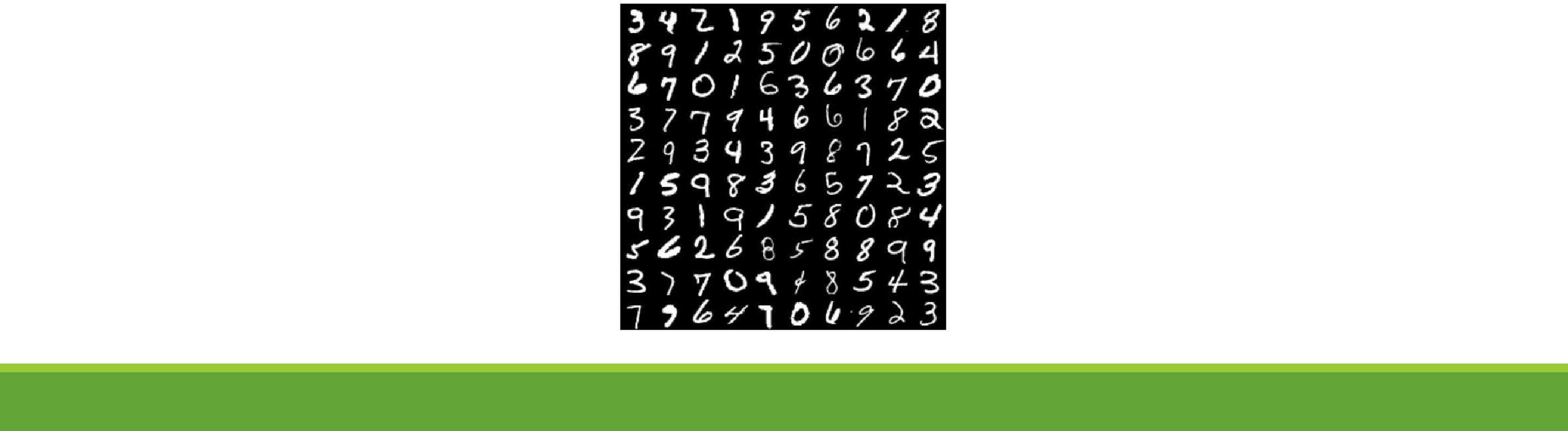
N binary N binary

maps maps



BinaryNet – Training and Testing

* We used and adapted some already available Keras code [5] to train a network on the MNIST dataset [6]
* 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.

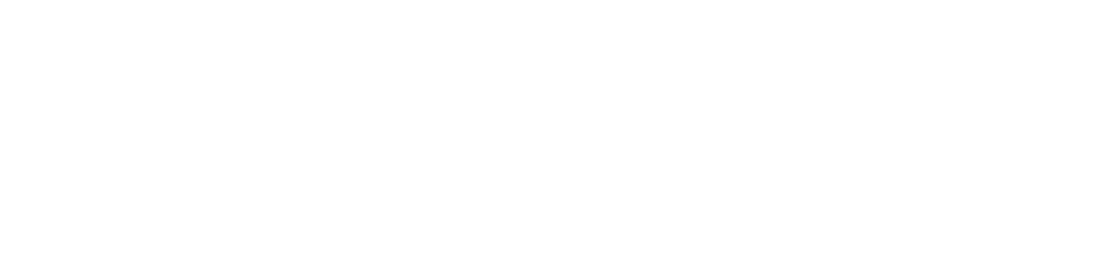
About 85% of the total weights belong to the first fully connected layer.



BinaryNet – Training and Testing

* 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 must go 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

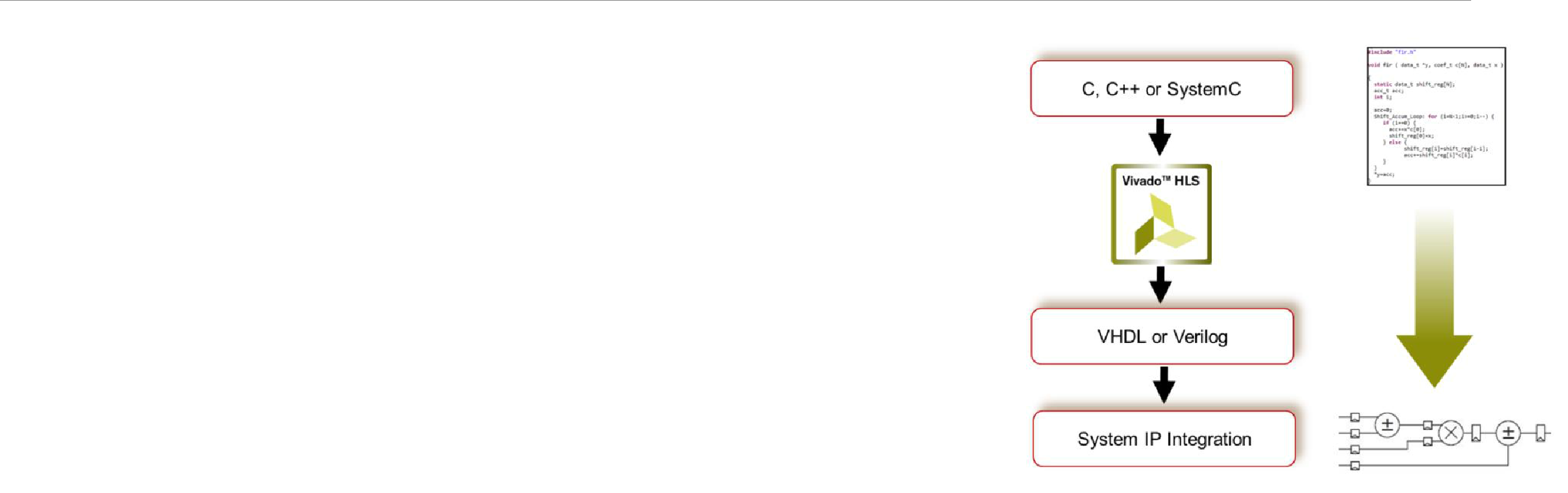


BinaryNet Implementation

* Our target architecture is the Xilinx Zynq processing unit ZC-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 pragams, that allow the programmer to control resource usage, timing requirements and the architecture implementation [7].
* 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)

* Other works [3, 4] exploit the power of High-Level Synthesis in order to implement a BinaryNet on a FPGA device



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

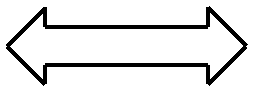


BinaryNet – Considerations for FPGA implementation

* 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** | **WEIGHT** | **PRODUCT** |
| **VALUE** | **VALUE** |  |
|  |  |  |
| 1 | 1 | 1 |
|  |  |  |
| 1 | -1 | -1 |
|  |  |  |
| -1 | 1 | -1 |
|  |  |  |
| -1 | -1 | 1 |
|  |  |  |

|  |  |  |
| --- | --- | --- |
| **INPUT** | **WEIGHT** | **XNOR** |
| **VALUE** | **VALUE** |  |
|  |  |  |
| 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 the 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 Zyng 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 and 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

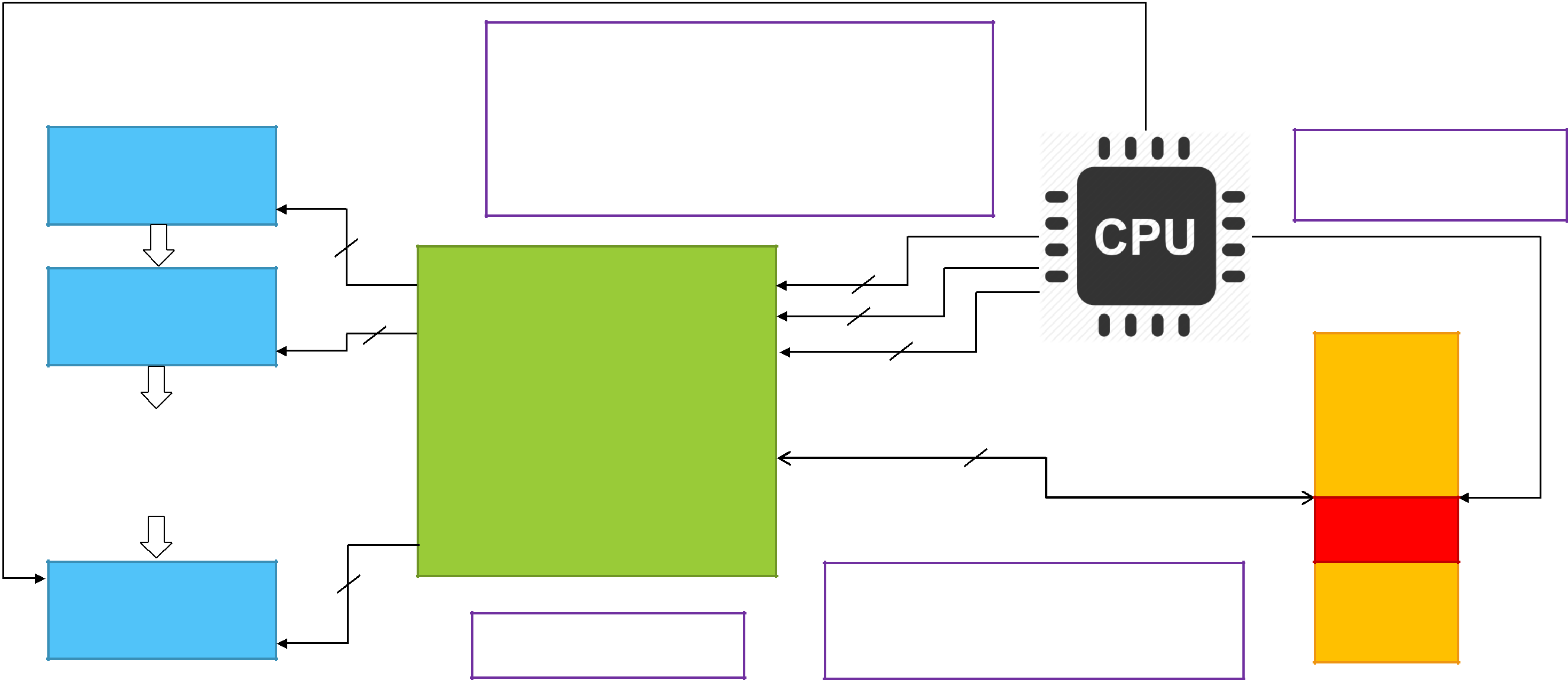
Disadvantages of the first approach over the alternative architecture:

* 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



 WEIGHT\_INTR

LAYER\_0

WEIGHT\_CHANNEL

**64**

2a. CPU wakes WEIGHT\_MUX and sends the index of

the layer that needs to be programmed, the number of

64bit words that the layer expects to receive and the

address where the weights need to be read in DDR 2b. CPU sends an interrupt to the designated layer. The layer starts listening on the weight channel port waiting for new weights

1. CPU saves weights of the designated layer on DDR (PS) sequentially

 WEIGHT\_INTR

LAYER\_1

WEIGHT\_CHANNEL

|  |
| --- |
| … |

WEIGHT\_INTR

LAYER\_M

WEIGHT\_CHANNEL

**64**

**64**

WEIGHT\_CHANNEL\_0 LAYER\_INDEX

NUM\_64\_WORDS

WEIGHT\_CHANNEL\_1

WEIGHT\_ADDRESS

WEIGHT\_MUX

|  |
| --- |
| … |

AXI\_MASTER\_DDR

WEIGHT\_CHANNEL\_M

1. The process is repeated sequentially for all layers



**32**

**64**

1. WEIGHT\_MUX reads the weights (using burst reads and buffers through HP ports) and sequentially sends each 64 bit packet to the designated layer (using LAYER\_INDEX)

RAM

WEIGHTS

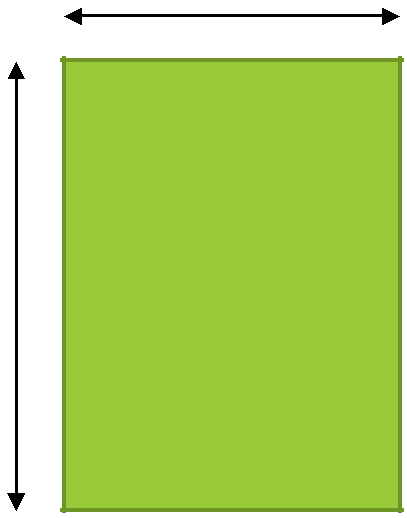
Fully connected layers



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.

OUTPUT\_DIM



|  |  |
| --- | --- |
| INPUT\_DIM | OUTPUT\_DIM |



|  |  |
| --- | --- |
| 1 | I |



|  |  |  |
| --- | --- | --- |
| X | DIM |  |
| INPUT\_ |  |
|  |  |

|  |  |
| --- | --- |
| = 1 | O |



W

= I ×

\_

=

=0



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 otuput value as an accumulator that at each iteration takes the sum between the i-th input value and the corresponding weight

double intput\_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 [-INPUT\_DIM, INPUT\_DIM]. With values that are 0 or 1, the range becomes [0, INPUT\_DIM] (only additions are performed -> simple counters on FPGA!)
* 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 acc >= 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 OUTPUT\_DIM / 64 if OUTPUT\_DIM is divisible by 64, OUTPUT\_DIM / 64 + 1 otherwise (to store the remaining most significant bits)
  + Example with OUTPUT\_DIM == 150 (64 + 64 + 22):

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| (long\*) addr | (long\*) addr + 1 | (long\*) addr + 2 | |  |  |
|  |  |  |  |  |  |
| W[0][0::63] | W[0][64::127] | W[0][128::149] | 42 unused bits |  |  |
|  |  |  |  | … |  |
|  |  |  |  |  |
| W[1][0::63] | W[1][64::127] | W[1][128::149] | 42 unused bits |  |
|  |  |  |  |  |  |



(long\*) addr + 3 (long\*) addr + 4 (long\*) addr + 5



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 log2( + 1)
* 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 omniscent 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

