Lab on HW-SW digital systems codesign
SEL B.Sc. Curriculum (Spring Semester)
Embedded Systems Laboratory
EPFL-STI-IEL

Exercises Session 3
Out: 2025-03-06

Accelerating a CNN-based application to discriminate dogs and cats

1 Introduction

Convolutional neural networks (CNNs) are commonly used to perform tasks such as image classification. In this example, we work with a CNN that classifies images of dogs and cats, that is, the application will read an input image and process it using a CNN that will assign a probability of it being a photograph of a dog or a cat. The output of the CNN is a value in the range $[0, 0.5)$ for the label “cat,” and a value in the range $(0.5, 1.0]$ for the label “dog.”

The execution of the application to perform *one single inference* on the Pynq board takes about 28 s. The goal of this and the following sessions is to reduce the execution time of the application developing one or more hardware accelerators using HLS.

1.1 Basics of convolutional neural networks

A CNN is a type of artificial neural network that relies on convolution operations to identify “features” in the input data. In a typical CNN structure, one convolution operation is followed by an activation function (e.g., a ReLU[1]) and, optionally, a pooling layer to reduce the dimensionality of the data. Multiple layers of convolution-ReLU-pooling operations are chained one after the other. At the end of the process, the final activation map can be used, for example, with a fully connected layer to implement a classifier based on probabilities. Figure 1 shows a typical CNN structure.

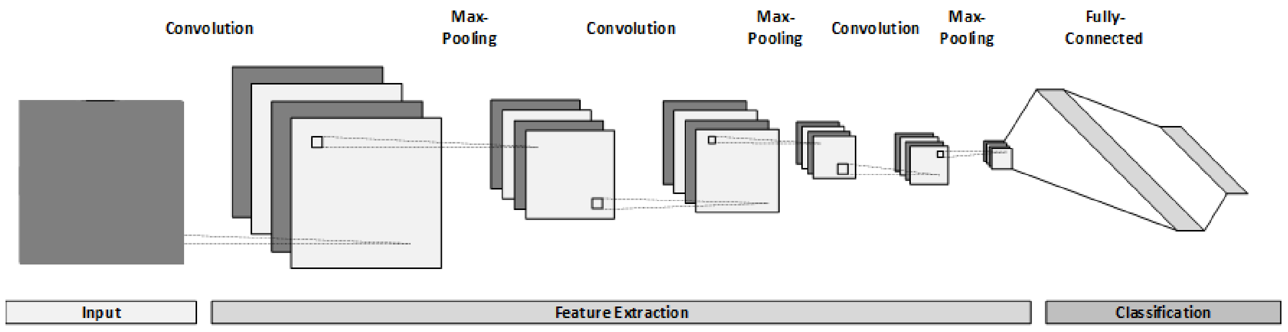


Figure 1: Typical CNN structure with three convolutional layers. Each pooling layer reduces the dimensionality (e.g., width and height) of the data.

[1]The rectified linear unit (ReLU) function returns x if $x > 0$; otherwise, it returns 0.

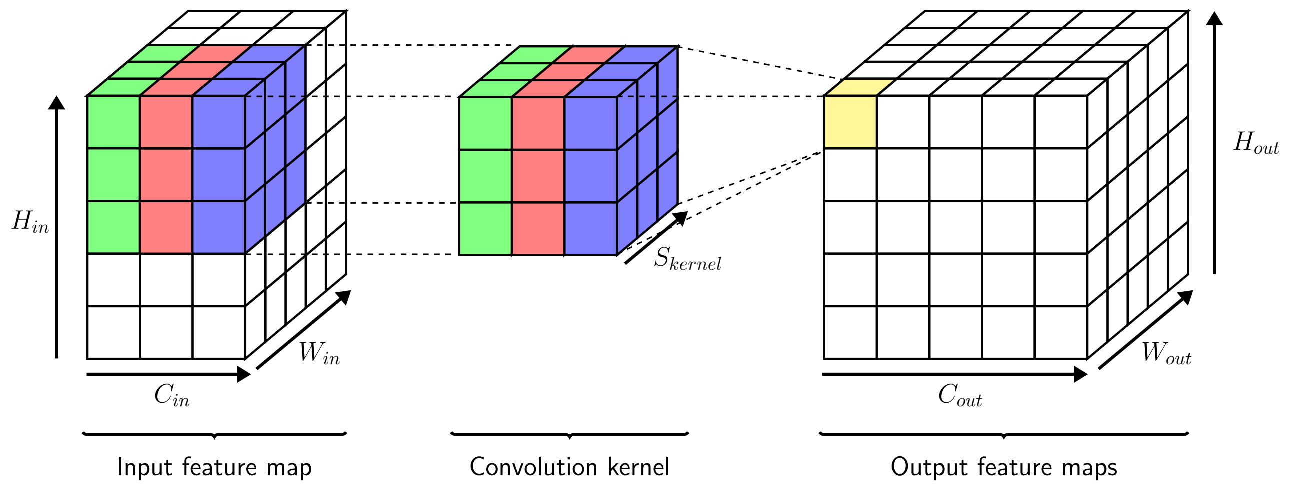


Figure 2: Example of convolutional layer. The input feature map can contain multiple channels (3 in this figure, one for each R, G, B color channel). Each filter, trained to detect specific features, has a set of coefficients for each channel in the input — in this example, there are 3×3 coefficients for each input feature map, for a total of 27 coefficients *per filter*. Each filter produces one output feature map after it is slided over the complete input feature maps.

The input to a convolutional layer consists of one or several channels. For example, a color image may have three channels (e.g., R, G and B). The layer has a set of filters, each of which is designed to detect specific features in the input. Typical filter sizes are 3×3, 5×5, 7×7 or 11×11. A filter of size 3×3 has 9 coefficients. If the input data has multiple channels, the filter is applied to each of the channels in the same position. Therefore, if the input data has N channels, then the filter may contain $3 \times 3 \times N$ coefficients — that is, the filter is applied over a region of each channel, using different coefficients for each of the channels.

During processing, the filter is slided over the input data with a certain stride (e.g., 1, 2, ...). Every time the filter is applied to all the channels of the input, the results computed for each channel are accumulated to produce the value of the output data at the current position.

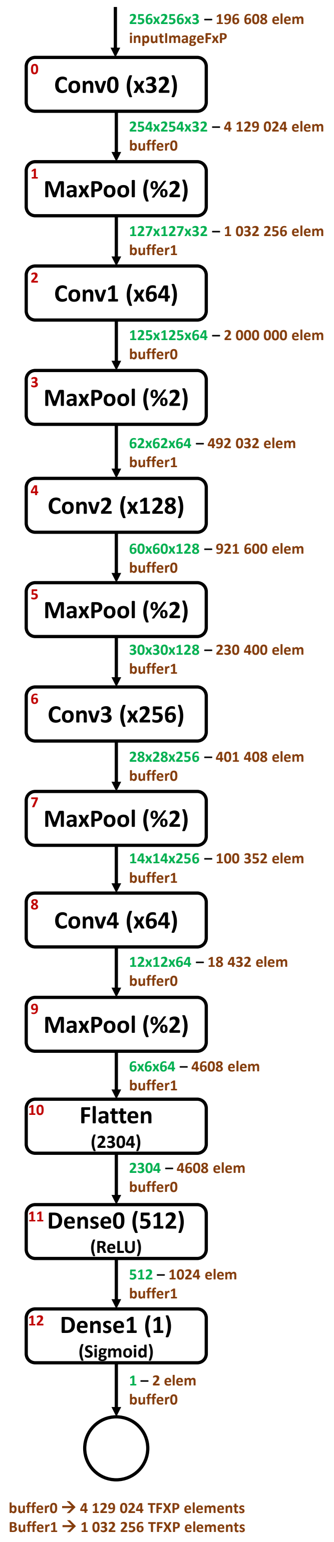
Additionally, a convolutional layer can consist of multiple parallel filters, each of them trained to identify specific features in the input. For example, one filter can be trained to detect curved shapes, one for straight lines, other to detect adjacent red-blue pixels, ... If a layer has M different filters, it will produce M output *feature maps* that will be interpreted as M input channels by the following layer. In that way, the number of output filters (i.e., output feature maps) of layer i is the number of input channels (i.e., input feature maps) of layer $i+1$. Figure 2 shows the complete process.

Convolutional layers typically account for the major part of computation time in a CNN. Therefore, they are often the prime target for acceleration.

1.2 CNN layer structure

The proposed CNN consists of five convolutional layers, each followed by a max pooling layer. Then, it has two fully connected layers ("dense") and a final sigmoid function to produce a single output classification value.

The following figure shows the CNN structure and the different parameters of each layer:



"Conv0" is a convolutional layer with 32 (output) filters, each of them operating on the 3 (input) channels (R, G, B) of the image. "Conv1" has 64 (output) filters, each of them processing its 32 (input) channels.

"Dense0" is a fully-connected layer with 512 neurons and 2304 inputs; it includes a ReLU activation function. On its side, "Dense1" has 1 neuron with 512 inputs; it uses a sigmoid activation function to produce the final result of the CNN. The "Flatten" layer is used in TensorFlow simply to reorder the parameters for the dense layers.

The C++ implementation of the CNN uses two buffers in a ping-pong manner, alternating them as input or output of successive layers. Each buffer has to be large enough to accommodate its largest possible contents. The previous figure shows the maximum size of each buffer in the proposed structure, in terms of number of elements of type `TFXP` — i.e., since `TFXP` is defined as `int32_t` in the code, multiply by 4 to know the actual size of the buffers in bytes.

1.3 Running the code

The files for the complete SW implementation of the project are available in Moodle. Download and use them as starting point of your work. The project includes the parameters of the complete CNN model in the `model/` folder. The CNN has been trained using TensorFlow with Keras and using a dataset of dog and cat images from `kaggle.com`. A set of sample images is provided in `images/`, with 100 images of dogs and 100 images of cats, labeled by filename. In the main folder of the project, there are also two individual test images for a cat and a dog, both in the planar RGB format[2] used by the application and in a PNG that can be opened with any standard tool. The planar images can also be visualized in GIMP if they are renamed to `*.data`. GIMP will show a pop-up window asking for the image format. In this case, the image is in "Planar RGB" format with a dimension of 256 × 256 pixels:

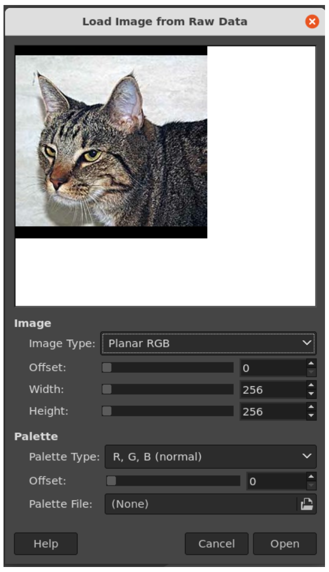


Image
Image Type: Planar RGB
Offset: 0
Width: 256
Height: 256
Palette
Palette Type: R, G, B (normal)
Offset: 0
Palette File: (None)

The accuracy of the original classification application, and later on of the developed accelerator(s), can be verified using the script `runAll.sh` . This script runs the CNN-based application on all the images in the `images/` folder and outputs global statistics. The original execution in SW, which the execution with our accelerator(s) must match, produces this output:

```
xilinx@pynq:~/dogs_cats/cnnSolver$ ./runAll.sh
images/dog.9400.jpg.rgba.planar
...
images/dog.9499.jpg.rgba.planar
images/cat.9400.jpg.rgba.planar
...
images/cat.9499.jpg.rgba.planar

Positive dogs: 87
Negative dogs: 13

Positive cats: 84
Negative cats: 16
```

We can also represent this information as a "confusion table":

[2]In planar RGB format, all the R values of the pixels are stored first, then all the G values are stored, and, finally, all the B values are stored together; this is as opposed to a "normal" RGB, RGB, ... distribution in which the three channel values for each pixel are stored one complete pixel after the other. The pixel values are stored in the range [0, 255].

	Real Dog	Real Cat
Inferred Dog	87 %	16 %
Inferred Cat	13 %	84 %

The result of each inference is stored in the files `outputDogs.txt` and `outputCats.txt`, for dogs and cats respectively. These files can be used to verify the execution on different images during the debugging process of the accelerator.

1.4 Code organization

The CNN-based application for classification of dog and cat images is structured into five files:

1. `cnn.h`, `cnn.cpp` These files implement the software version of each of the CNN layers. It contains the functions `Conv2D()`, `AddBiases()`, `ReLU()`, `MaxPool()`, `Sigmoid()`, `Dense()`, and `Flatten()`.
   To integrate, for example, a convolution accelerator into the application code, substitute `Conv2D()` with your implementation of `Conv2D_HW()`. Additionally, since a convolution accelerator will very likely also include the implementation of the bias and ReLU operations, these two functions will have to be removed from the SW version.
   All these functions work on fixed-point (FxP) values based on the user-defined type `TFXP`.
2. `model.h`, `model.cpp` These two files provide the required data types and auxiliary functions to load the model and the input image into memory.
   The model is stored on disk in the folder `model/`, in two separate files for each convolutional layer: one for the convolution coefficients (weights), another for the bias (one single value per output filter). All the weights are stored in floating-point (FP); hence, they have to be converted to FxP before they can be used in the application. The main application loads the model using `LoadModelInFxp()`.
   Input images are stored in planar RGB format, with pixel values in the range [0, 255]. The application loads the images and converts the pixel values to FxP values in the range [0, 1].
3. `cnnSolver.cpp` This is the main file of the SW CNN-based classification application. It loads the model and the input images, converts them into FxP, and calls the inference function.
   The application uses one buffer to store the pixel values in [0, 255] format (`inputImage`) and one buffer to store the pixel values in [0, 1] FxP format (`inputImageFxp()`). Since the size of the input image is fixed (to 256×256 pixels), these buffers are allocated statically. In contrast, the model weights are stored in an array of vectors (`weights[]` and `biases[]`). These two identifiers are a vector of pointers to dynamically allocated vectors. That is, `weights[0]` points to a dynamically-allocated vector that contains the weights of layer 0, `weights[1]` points to a dynamically-allocated vector that contains the weights of layer 1, etc. In that way, the size of each vector can be adjusted to the exact length of the vector itself.
   Intermediate results, i.e., *activations*, which are the output of one layer and become the input of the next layer, are stored using just two vectors. We use them in a ping-pong fashion, i.e., they switch the role of output and input for each consecutive layer. Therefore, their size has to be enough to accommodate the largest possible input or output that will be assigned to each of them. Refer to the figure in Section 1.2 to see how their respective maximum sizes are calculated.

2 Profiling: What functionality should be accelerated?

The source code of the C++ application is instrumented to show the execution time of every CNN layer. On the Pynq board, a typical execution looks like this:

```
xilinx@pynq:~/dogs_cats/cnnSolver$ ./cnnSolver dog.9499.jpg.rgba.planar
OUTPUT: 0.93905449 --> DOG
Conv 0 --> 1747935637 ns (1.748 s)
Conv 1 --> 8891987938 ns (8.892 s)
Conv 2 --> 8470657030 ns (8.471 s)
Conv 3 --> 7931560320 ns (7.932 s)
Conv 4 --> 794392039 ns (0.794 s)
MaxPool 0 --> 24272126 ns (0.024 s)
MaxPool 1 --> 10338197 ns (0.010 s)
MaxPool 2 --> 5494031 ns (0.005 s)
MaxPool 3 --> 2519431 ns (0.003 s)
MaxPool 4 --> 91973 ns (0.000 s)
Dense 5 --> 25825898 ns (0.026 s)
Dense 6 --> 12707 ns (0.000 s)
Total Conv time: 27836532964 ns (27.837 s) 99.8 %
Total MaxPool time: 42715758 ns (0.043 s) 0.2 %
Total Dense time: 25838605 ns (0.026 s) 0.1 %
Total Flatten time: 13662 ns (0.000 s) 0.0 %
Total Sigmoid time: 140661 ns (0.000 s) 0.0 %
Total time: 27905241650 ns (27.905 s) 100.0 %
```

The execution takes 27.9 s and the result is that the image is identified as a dog. Since all the input images have the same dimensions and the computation is data-independent, the execution time for any given image is approximately the same.

► **Question:** *Given the profiling results obtained* by you*, which parts of the application should be optimized using a hardware accelerator?*

The goal of this exercise is to reduce the inference time down from the original* $\sim$ 28 s *per image using a HW accelerator implemented on the PL side of the FPGA.

3 Task: Design of a convolution accelerator and integration in the complete application

The convolution operation is computationally intensive and offers opportunities for parallelism. These two characteristics make it a good candidate for acceleration.

3.1 Operations

3.1.1 Convolution

For the implementation of the convolution accelerator, let's assume that the filter size is always 3×3, and that the filter stride (the sliding step) is 1. Since the filter size is fixed to 3×3 and the stride is 1, the width and height of the output images will be 2 pixels less than for the inputs.

The accelerator will receive as parameters (in registers):

- The input data, as a set of 2D images, organized in multiple channels.
- A pointer to the output buffer, organized in multiple channels.
- The filter coefficients. For N input channels and M output filters, the coefficients will be organized in an array of $M \times N \times 3 \times 3$.
- Scalar parameters to indicate the input width and height, the number of input channels and the number of output filters.

The following fragment of code is used to implement the convolution functionality in the software application:

```
const uint32_t DECIMALS = 20;
typedef int32_t TFXP;     // Parameters and activations
typedef int64_t TFXP_MULT;// Intermmediate results of multiplications

inline TFXP FXP_Mult(TFXP a, TFXP b, uint32_t decimalBits = DECIMALS)
{
  //return a*b;
  TFXP_MULT res = (TFXP_MULT)a * (TFXP_MULT)b;
  res = res >> decimalBits;
  return res;
}

void Conv2D(TFXP *input, TFXP * output, TFXP * filters,
  uint32_t numFilters, uint32_t numChannels,
  uint32_t inputWidth, uint32_t inputHeight,
  uint32_t convWidth, uint32_t convHeight)
{
  for (uint32_t iFilter = 0; iFilter < numFilters; ++ iFilter) {
    for (uint32_t y = 0; y < (inputHeight-2); ++y) {
      for (uint32_t x = 0; x < (inputWidth-2); ++ x) {
        TFXP acc;
        acc = 0;
        for (uint32_t iChannel = 0; iChannel < numChannels; ++ iChannel) {
          for (uint32_t cy = 0; cy < convHeight; ++ cy) {
            for (uint32_t cx = 0; cx < convWidth; ++cx) {
              //acc += filters[iFilter][iChannel][cy][cx] * input[iChannel][y+cy][x+cx];
              TFXP v, f;
              f = *(filters + iFilter*numChannels*convHeight*convWidth + iChannel*convHeight*convWid
              v = *(input + iChannel*inputWidth*inputHeight + (y+cy)*inputWidth + (x+cx));
              acc += FXP_Mult(f, v, DECIMALS);
            }
          }
        }
        //output[iFilter][y][x] = acc;
        *(output + iFilter * (inputHeight-2)*(inputWidth-2) + y*(inputWidth-2) + x) = acc;
      }
    }
  }
}
```

This implementation uses FxP numbers — check Appendix A for more details about using FxP arithmetic in HLS descriptions. To test the peripheral initially, isolated from the rest of the application, it is possible to use an input buffer with random data and compare the outputs produced by the SW implementation and by your accelerator. This implementation also transforms the multi-dimensional arrays of the input data into pointers to flattened (contiguous) buffers of data. For reference, the original definition of the input vectors could have been like this:

```
void Conv(TFXP input[NUM_CHANNELS][INPUT_HEIGHT][INPUT_WIDTH],
  TFXP output[NUM_FILTERS][OUTPUT_HEIGHT][OUTPUT_WIDTH],
  TFXP filters[NUM_FILTERS][NUM_CHANNELS][CONV_HEIGHT][CONV_WIDTH],
  ...
```

The primary goal of this exercise is to focus on the correct functionality of the peripheral and integration on the complete application — it is not necessary to obtain an efficient implementation yet.

> ► **Question:** *Compare the execution time of the algorithm running as SW on the ARM Cortex-A9 processors with the execution time of your peripheral. Using Table 1 as reference, create a table with the execution time of each of your solutions with their respective use of resources.*

Description	Width	Height	Channels	Filters	Time (ms)	LUTs	FFs	BRAMs	DSPs
SW	256	256	16	32	x	n/a	n/a	n/a	n/a
Initial solution	256	256	16	32	x	x	x	x	x

Table 1: Example of results table: "Execution time and use of resources of each of our solutions for the CNN accelerator." (n/a stands for "not applicable")

3.1.2 Bias

The bias is a value, defined per output filter, that is added to every output pixel. After the value of the pixel is calculated via the convolution operation, the bias is added to the result to produce the final pixel value. A convolutional layer with 32 output filters has 32 bias values, one associated to each output filter. Implementing this operation as the output values are computed saves significant memory bandwidth, since otherwise every output pixel produced by a convolutional layer has to be read and updated after the convolution.

To implement the bias, the accelerator has to receive a pointer to the weights of the layer, plus another pointer to the bias values.

3.1.3 ReLU

Convolutional layers are usually followed by a non-linear activation function. A common function is the ReLU, which simply copies positive values, and sets to zero any negative values:

$$f(x) = \begin{cases} x & \text{if } x \geq 0 \\ 0 & \text{otherwise} \end{cases}$$

The ReLU operation is applied to every output pixel, after adding the bias of the corresponding filter. Similarly to the previous case, implementing this operation in the accelerator before the pixels are written to the DRAM saves one read and one write of the complete output data. To flexibly incorporate the ReLU function, the accelerator needs an additional parameter that indicates whether the function should be applied or not.

3.1.4 FxP 32-bit vs. 16-bit

FxP is often faster and uses less resources than FP arithmetic. For that reason, CNNs are often quantized to use FxP arithmetic during inference. Our software CNN application uses a FxP representation of 32 bits with 20 decimal bits, as shown in the following fragment of code from `model.h`:

```
const uint32_t DECIMALS = 20;
typedef int32_t TFXP;     // Model parameters and activations
typedef int64_t TFXP_MULT;// Intermmediate results of multiplications
typedef int32_t TFXP_ACC; // Convolution accumulators
```

In order to obtain the correct results, the multiplications have to be performed in double the bitwidth than the parameters and activations.[3] In cases in which the number of input channels is large, the accumulators used during the computation may also require a larger bitwidth; however, in our case, 32 bits are enough.

▶ **Observation:** *You can explore different implementations and use of resources: Inference on the provided version of the trained CNN can also be done using a width of 16 bits for the parameters, activations and accumulators, with 14 decimal bits; and 32 bits for the multiplications. However, since the model has been trained directly with floating point numbers (without quantization during training), the accuracy of the network will be severely reduced.*

3.2 Testing the accelerator

3.2.1 Testing in HLS before synthesis

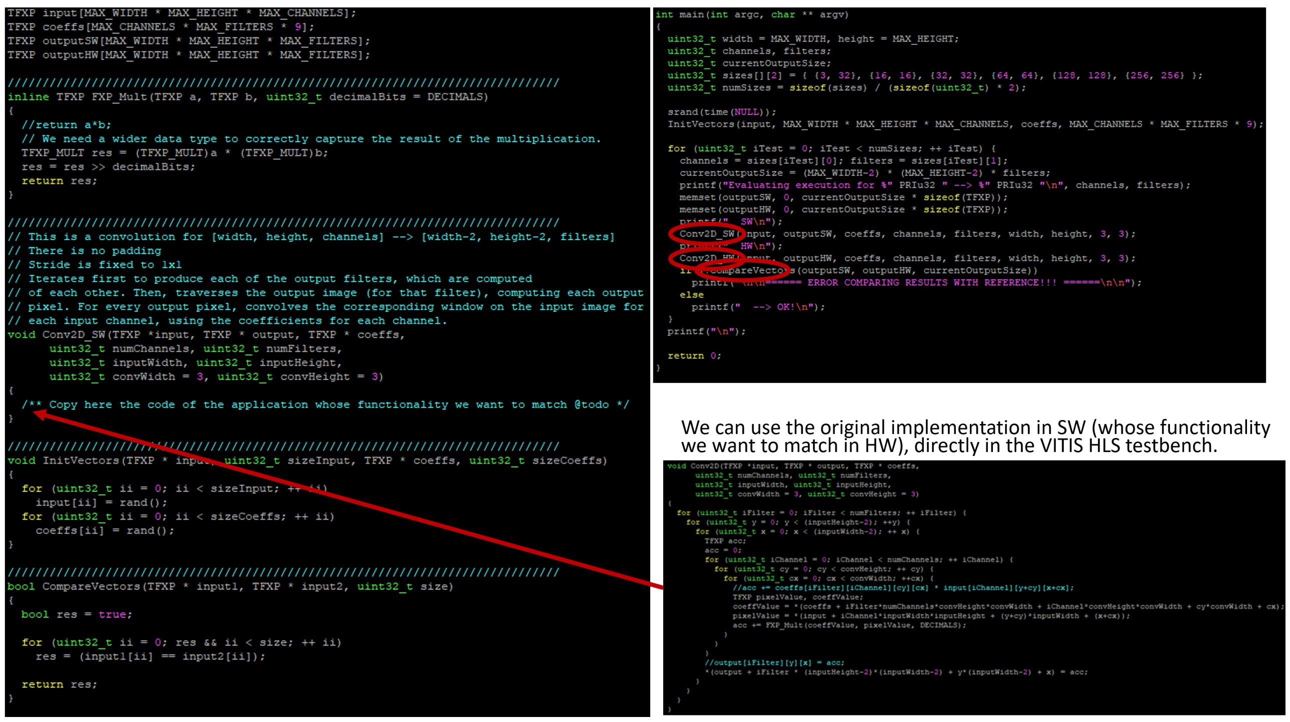
Building a testbench for use with Vitis HLS before synthesis will speed up enormously the debugging process, since correct algorithmic behavior can be verified before starting the lengthy synthesis process. The testbench can be integrated in the Vitis HLS work flow so that, every time that we introduce any modification in the HW description, its correct functionality will be verified before the synthesis process.

In Moodle, you will find the skeleton for the simulation process. During C simulation, we can take advantage of the fact that the HW description is actually written in C++ and can hence be compiled and executed on the host PC. Our testbench has to define a `main()` function that will be the entry point of the testbench. Then, it can call the top function of the accelerator, just as if it were a normal function call in a software program. The result of the accelerator function, which may contain complex transformations to create an efficient architectural description in hardware, can be compared with the result of the original software function from the initial application. Figure 3 shows how the pieces of the example (from Moodle) fit together in Vitis HLS.

3.2.2 Testing in the Pynq board after bitstream generation

Once the bitstream is generated, we can test the functionality of the accelerator in a simplified testbench, rather than in the complete CNN application. The `Conv2D()` function is executed by the processors using FxP arithmetic, which is actually implemented using integer arithmetic (addition and multiplication with shift). This is our ground truth.

[3]The activations are the outputs of each layer, which become the inputs of the next layer.



```
TFXP input[MAX_WIDTH * MAX_HEIGHT * MAX_CHANNELS];
TFXP coeffs[MAX_CHANNELS * MAX_FILTERS * 9];
TFXP outputSW[MAX_WIDTH * MAX_HEIGHT * MAX_FILTERS];
TFXP outputHW[MAX_WIDTH * MAX_HEIGHT * MAX_FILTERS];

///////////////////////////////////////////////////////////////////////////////
inline TFXP FXP_Mult(TFXP a, TFXP b, uint32_t decimalBits = DECIMALS)
{
  //return a*b;
  // We need a wider data type to correctly capture the result of the multiplication.
  TFXP_MULT res = (TFXP_MULT)a * (TFXP_MULT)b;
  res = res >> decimalBits;
  return res;
}

///////////////////////////////////////////////////////////////////////////////
// This is a convolution for [width, height, channels] --> [width-2, height-2, filters]
// There is no padding
// Stride is fixed to 1x1
// Iterates first to produce each of the output filters, which are computed
// of each other. Then, traverses the output image (for that filter), computing each output
// pixel. For every output pixel, convolves the corresponding window on the input image for
// each input channel, using the coefficients for each channel.
void Conv2D_SW(TFXP *input, TFXP * output, TFXP * coeffs,
      uint32_t numChannels, uint32_t numFilters,
      uint32_t inputWidth, uint32_t inputHeight,
      uint32_t convWidth = 3, uint32_t convHeight = 3)
{
  /** Copy here the code of the application whose functionality we want to match @todo */
}

///////////////////////////////////////////////////////////////////////////////
void InitVectors(TFXP * input, uint32_t sizeInput, TFXP * coeffs, uint32_t sizeCoeffs)
{
  for (uint32_t ii = 0; ii < sizeInput; ++ ii)
    input[ii] = rand();
  for (uint32_t ii = 0; ii < sizeCoeffs; ++ ii)
    coeffs[ii] = rand();
}

///////////////////////////////////////////////////////////////////////////////
bool CompareVectors(TFXP * input1, TFXP * input2, uint32_t size)
{
  bool res = true;

  for (uint32_t ii = 0; res && ii < size; ++ ii)
    res = (input1[ii] == input2[ii]);

  return res;
}
```

```
int main(int argc, char ** argv)
{
  uint32_t width = MAX_WIDTH, height = MAX_HEIGHT;
  uint32_t channels, filters;
  uint32_t currentOutputSize;
  uint32_t sizes[][2] = { {3, 32}, {16, 16}, {32, 32}, {64, 64}, {128, 128}, {256, 256} };
  uint32_t numSizes = sizeof(sizes) / (sizeof(uint32_t) * 2);

  srand(time(NULL));
  InitVectors(input, MAX_WIDTH * MAX_HEIGHT * MAX_CHANNELS, coeffs, MAX_CHANNELS * MAX_FILTERS * 9);

  for (uint32_t iTest = 0; iTest < numSizes; ++ iTest) {
    channels = sizes[iTest][0]; filters = sizes[iTest][1];
    currentOutputSize = (MAX_WIDTH-2) * (MAX_HEIGHT-2) * filters;
    printf("Evaluating execution for %" PRIu32 " --> %" PRIu32 "\n", channels, filters);
    memset(outputSW, 0, currentOutputSize * sizeof(TFXP));
    memset(outputHW, 0, currentOutputSize * sizeof(TFXP));
    printf("  SW\n");
    Conv2D_SW(input, outputSW, coeffs, channels, filters, width, height, 3, 3);
    printf("  HW\n");
    Conv2D_HW(input, outputHW, coeffs, channels, filters, width, height, 3, 3);
    if (!CompareVectors(outputSW, outputHW, currentOutputSize))
      printf("\n\n====== ERROR COMPARING RESULTS WITH REFERENCE!!! ======\n\n");
    else
      printf("  --> OK!\n");
  }
  printf("\n");

  return 0;
}
```

We can use the original implementation in SW (whose functionality we want to match in HW), directly in the VITIS HLS testbench.

```
void Conv2D(TFXP *input, TFXP * output, TFXP * coeffs,
      uint32_t numChannels, uint32_t numFilters,
      uint32_t inputWidth, uint32_t inputHeight,
      uint32_t convWidth = 3, uint32_t convHeight = 3)
{
  for (uint32_t iFilter = 0; iFilter < numFilters; ++ iFilter) {
    for (uint32_t y = 0; y < (inputHeight-2); ++y) {
      for (uint32_t x = 0; x < (inputWidth-2); ++ x) {
        TFXP acc;
        acc = 0;
        for (uint32_t iChannel = 0; iChannel < numChannels; ++ iChannel) {
          for (uint32_t cy = 0; cy < convHeight; ++ cy) {
            for (uint32_t cx = 0; cx < convWidth; ++cx) {
              //acc += coeffs[iFilter][iChannel][cy][cx] * input[iChannel][y+cy][x+cx];
              TFXP pixelValue, coeffValue;
              coeffValue = *(coeffs + iFilter*numChannels*convHeight*convWidth + iChannel*convHeight*convWidth + cy*convWidth + cx);
              pixelValue = *(input + iChannel*inputWidth*inputHeight + (y+cy)*inputWidth + (x+cx));
              acc += FXP_Mult(coeffValue, pixelValue, DECIMALS);
            }
          }
        }
        //output[iFilter][y][x] = acc;
        *(output + iFilter * (inputHeight-2)*(inputWidth-2) + y*(inputWidth-2) + x) = acc;
      }
    }
  }
}
```

Figure 3: Example.

We can compare the output of the accelerator with the output of the SW function, for the same input vectors. To do this, we can create a function named `Conv2D_HW()`, which calls the accelerator, and compare the results generated by both versions. There is an example in Moodle that can serve as guide in this task.

3.3 Integration in the application

The model itself is solved in the function `Inference()`, which returns a value in the range [0, 1]. *To introduce your HW accelerator, substitute the calls to `Conv2D()` inside the `Inference()` function in the file `model.cpp`, using instead a function `CConv2DDriver::Conv2D_HW()` provided by your accelerator driver class.* Then, modify the memory allocation functions to use the DMA-compatible functions provided by the driver class of your accelerator (see Section 3.3.1).

Filter bias and ReLU activation functions are normally subsumed in the convolutional layers. In this way, additional movements of data between the memories and the processing elements are avoided. Therefore, remove the calls to `AddBiases()` and `ReLU()` when your accelerator implements these additional operations.

The accelerated application using the accelerator must produce the same classification over the 200 images than the original software-only implementation. That is, the confusion matrix for the software and accelerated versions must be identical. This applies to the classification outcome itself, into the “dog”/“cat” categories; the actual output values of the network (after the sigmoid function) can vary slightly. This will likely happen if the rounding modes for FxP arithmetic are not exactly the same in HLS than in the C++ implementation.

3.3.1 Memory allocation for use with the accelerator

The application loads the weights in two steps: First, it loads the FP weight values into memory; then, it converts the weights into FxP format. The FP values are only processed by SW in the CPU and their memory is immediately freed upon conversion. Since these values are never shared with the accelerator, they are allocated and freed using the normal `malloc()` and `free()` functions. In contrast, the FxP values are used by the accelerator during the computation of the different layers.

Therefore, these vectors need to be allocated and freed in a way compatible with direct memory access (DMA). The application allocates memory space for the FxP weights in the functions `ConvertWeightsToFxp()` and `ConvertBiasesToFxp()` using `malloc()`. We need to substitute these calls to `malloc()` and `free()` by calls to the method `AllocDMACompatible()` from the accelerator driver class.

> ▶ **Observation:** *In the next sessions of the course we will explore how these functions work and why they are necessary.*

> ▶ **Question:** *Why don't we use* `AllocDMACompatible()` *to allocate all the vectors, avoiding the distinction between normal and DMA-compatible memory allocations?*

Input images are stored in disk in RGB *planar* format, i.e., all the R values of the pixels are stored first, then all the G values are stored, and, finally, all the B values are stored together (in contrast with a "normal" RGB, RGB, ... distribution). However, the values are stored in the range [0, 255]. CNN models typically operate on values in the range [0, 1]. Therefore, the application loads the pixel values from disk, normalizes them to the range [0, 1], and converts them into FxP format. This is done in function `LoadImageInFxp()`. In the pure SW solver, both vectors are allocated statically in `cnnSolver.cpp` (since the input image size is fixed to 256×256). When we introduce the hardware accelerator, the vector for the FxP pixels of the input image has to be allocated and deallocated using the DMA-compatible function of the accelerator driver.

Finally, the vectors used to store intermediate activations in `cnnSolver.cpp` (`buffer0` and `buffer1`) are also used by the accelerator to store the output of a layer and to read the input of the next layer (i.e., they hold the layer activations). Therefore, these two vectors also have to be allocated using DMA compatible memory.

As a final note, do not forget to free all the DMA vectors allocated by your application, since they are global-system resources and are not freed automatically at the end of your application. If any vector remains allocated at the end of the execution, the application will display an error message as a reminder and it will still automatically free the vectors for you. However, if your application crashes, the DMA memory will not be freed and eventually the complete system will need to be rebooted.

3.4 Measuring performance with the accelerator

Using the parameters included in Table 1, calculate (approximately) the total amount of data that the accelerator would need to transfer to/from the main memory (reads and writes), and the total amount of multiply-accumulate (MAC) that your accelerator would need to compute. Adding a System ILA module to the design will help to analyze the data transfers performed by the accelerator. How many transactions does it perform? What data is being transferred in each transaction (the size in each dimension)? How can this transaction pattern to/from memory be improved?

In Vitis HLS, open the *Synthesis Summary Report* and look at the latency and iteration interval (II) of each loop. Use these numbers together with the parameters given in Table 1 to estimate the total number of cycles it takes to perform a convolution. How many cycles does the system take to perform a MAC operation? This number can be approximated by taking the total amount of cycles calculated before and dividing it by the total number of MACs performed. How many MACs do you think can be parallelized and what would be the impact on performance?

A FxP in HLS

CNNs are usually trained using FP numbers. However, FP arithmetic is expensive in terms of hardware area, latency and energy. Therefore, it is common to convert all the CNN parameters to some type of FxP representation in a process that is commonly known as "quantization." For example, we can quantize the weights into 8 bit FxP, and the values of the intermediate feature maps (*activations*) into 16 bit FxP. Furthermore, in many cases it is possible to obtain more aggressive quantizations without significant impact on accuracy. One important consideration is that, to ensure that the FxP representations maintain enough accuracy, the intermediate accumulators are typically stored in a larger datatype (e.g., a 32-bit FxP number).

► **Question:** *Why is a larger datatype normally used for the accumulation of intermediate results when computing convolutional layers using FxP arithmetic?*

For the execution in C++, we could use a specialized FxP library such as fixmath (`http://www.nongnu.org/fixmath/doc/`). However, since we only perform basic operations (additions, multiplications and comparisons), we can implement the arithmetic directly using macros and operating on native types. For example, using `uint16_t`, the addition of two fixed-point numbers can be performed with a normal addition operator, whereas the multiplication requires an additional shift-right step of the number of decimal bits in the representation.

A.1 FxP support in Xilinx HLS

Xilinx HLS includes support for FxP data types through the template classes `ap_ufixed<>` and `ap_fixed<>`:

```
#include "ap_fixed.h"

ap_[u]fixed<W,I,Q,O,N>
ap_fixed<16, 9> cnnWeights[SIZE];
```

Where the parameters W, I, Q, O and N are defined as in Table 2.

Value	Meaning
W	Word length in bits
I	Integer bits to the left of the (binary) point
Q	Quantization mode. Can be `AP_RND`, `AP_RND_ZERO`, `AP_RND_MIN_INF`, `AP_RND_INF`, `AP_RND_CONV`, `AP_TRN`, `AP_TRN_ZERO` – Check the documentation!
O	Overflow mode: `AP_SAT`, `AP_SAT_ZERO`, `AP_SAT_SYM`, `AP_WRAP`, `AP_WRAP_SM`
N	Number of saturation bits in overflow wrap modes

Table 2: Meaning of the parameters of the classes `ap_fixed<>` and `ap_ufixed<>`.

For example, if we have a FxP representation of $Q_{15.7}$, that is, 16 bits divided into sign bit, 8 bits for the integer part, and 7 bits for the decimal part. In HLS, we can use the type `ap_fixed<16, 9>`.[4]

A.2 Converting values between FxP and FP in HLS

The `ap_fixed<>` data type has an overloaded assignment operator and a method to convert to float:

```
ap_fixed<4,2> a = 1.25;
printf("%lf\n", a.to_float());
```

In addition, the binary representation in memory is equivalent to the one we would use in SW — at least, for standard sizes as `int8_t`, `int16_t`, `int32_t`. This is not documented as far as we know, so take this carefully! Using the conversion functions is a safer option.

[4]In the HLS type `ap_fixed<W, I>`, W represents the total number of bits, while I represents the number of integer bits (including sign bit, if applicable), whereas in $Q_{t.d}$, d represents the number of decimal bits. Therefore, $Q_{15.7}$ is equivalent to `ap_fixed<16, 9>`.

B Data formats and layout used in the application

A CNN solver needs to load the model data, i.e., the weights of the coefficients for each filter. Then, it has to load the input image, which can be composed of multiple channels (e.g., one for each R, G, B component of a color image). Every layer in the network has an input and an output buffer. Since the models we are using are sequential, we can use just two buffers in a ping-pong manner, alternating the role of input and output for each buffer at each layer. The size of the buffers has to be enough to accommodate the input and output data for the largest layer. The following pseudo-code illustrates how `buff[0]` and `buff[1]` alternate roles for each consecutive layer:

```
1.- Allocate buff[2][...]
2.- Load layer 1 coeffs
3.- Load input image into buff[0]
4.- Call Conv(buff[0], buff[1], coeffs)
8.- Load layer 2 coeffs
9.- Call Conv(buff[1], buff[0], coeffs)
10.- ...
```

B.1 Layout of coefficients

To simplify the work, we can assume that the model coefficients are stored in the file and in memory as `coeffs[FILTERS][CHANNELS][KSIZE]`, where `KSIZE` is the number of coefficients in the kernel, linearized (i.e., $3 \times 3 = 9$ coefficients in our case):

$F_0C_0k_0, F_0C_0k_1, F_0C_0k_2, F_0C_0k_3, F_0C_0k_4, F_0C_0k_5, F_0C_0k_6, F_0C_0k_7, F_0C_0k_8,$
$F_0C_1k_0, F_0C_1k_1, F_0C_1k_2, F_0C_1k_3, F_0C_1k_4, F_0C_1k_5, F_0C_1k_6, F_0C_1k_7, F_0C_1k_8,$
$\dots$
$F_0C_{n-1}k_0, F_0C_{n-1}k_1, F_0C_{n-1}k_2, F_0C_{n-1}k_3, F_0C_{n-1}k_4, F_0C_{n-1}k_5, F_0C_{n-1}k_6, F_0C_{n-1}k_7, F_0C_{n-1}k_8,$

$F_1C_0k_0, F_1C_0k_1, F_1C_0k_2, F_1C_0k_3, F_1C_0k_4, F_1C_0k_5, F_1C_0k_6, F_1C_0k_7, F_1C_0k_8,$
$\dots$
$F_1C_{n-1}k_0, F_1C_{n-1}k_1, F_1C_{n-1}k_2, F_1C_{n-1}k_3, F_1C_{n-1}k_4, F_1C_{n-1}k_5, F_1C_{n-1}k_6, F_1C_{n-1}k_7, F_1C_{n-1}k_8,$

$F_{m-1}C_0k_0, F_{m-1}C_0k_1, F_{m-1}C_0k_2, F_{m-1}C_0k_3, F_{m-1}C_0k_4, F_{m-1}C_0k_5, F_{m-1}C_0k_6, F_{m-1}C_0k_7, F_{m-1}C_0k_8,$
$\dots$
$F_{m-1}C_{n-1}k_0, F_{m-1}C_{n-1}k_1, \dots, F_{m-1}C_{n-1}k_8,$

where F_i is the (output) filter index, C_i is the input channel index, and k_i is the kernel coefficient. In this way, we can read all the coefficients together from disk — converting them from FP to FxP format — and call the accelerator passing a portion of the array, such as in `coeffs[iFilter]`.

B.2 Layout of image data

Similarly, we can assume that the images are stored in order `[CHANNEL][HEIGHT][WIDTH]` — i.e., the image of each channel is stored entirely and channels are consecutive. If we store the output of every filter consecutively, such as `[FILTER][HEIGHT][WIDTH]`, then we can chain layers directly without reshaping our arrays.

This memory organization is convenient for our exercise, but may not be the most efficient in all cases. In particular, for narrow input feature maps with many channels, it may be better to store the data in a different order.