Applied machine learning
2024
AI visual odometry

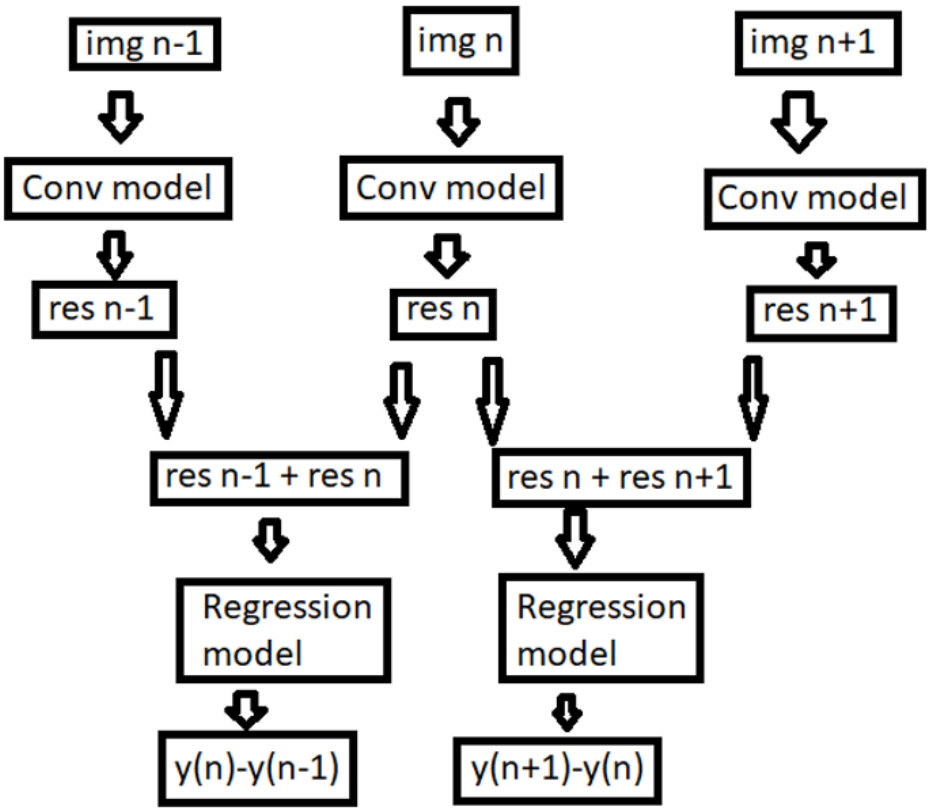
Arnas Serva
Tommi Tofthøj
Frej Scherfig

Visual odometry

Visual Odometry is the process of estimating a camera's position and orientation in 3D space by analyzing a sequence of images captured by the camera. It uses visual features in the images to track motion and reconstruct the path traveled, often used in robotics, autonomous vehicles, and augmented reality to navigate and understand environments without external positioning systems like GPS.

Model structure

The increased efficiency of this method comes from its incremental processing approach. Instead of recalculating parameters for both images at every step, the algorithm saves the parameters of the most recent image. This means that for each new step, only the parameters of the new image need to be computed, rather than processing both images again as in traditional algorithms. This reduces computational load, speeds up processing, and makes the method more suitable for real-time applications.



Project subpart: Data generation

Getting accurate positional data and matching it with images is difficult to do in real life. Because of this we looked towards Nvidia ISAAC. Nvidia ISAAC is a simulation tool which allows the user to control everything in a 3D space and generate photorealistic images. We used this to generate a 3D environment replicating a warehouse. In this warehouse we then moved the camera around and save images together with the current position of the camera.

Project subpart: Training model

After generating a dataset consisting of the images and positions while the camera was moving in the forward direction we paired the data so that the input to the model would be two images and then the output is the distance between them in the forward direction of the camera. The two convolutional parts of the model is a pretrained model we made ourselves and trained as an image classifier on another dataset created in Nvidia ISAAC. That dataset consisted of images of 6 different 3D objects in the warehouse environment. The dense layers were then removed from that model and the outputs from the convolutional layers of two instances of that model were then combined into being the input to some dense layers which then output the distance in the camera direction. This model was then trained on the full dataset with two images as the input and the distance between them as the output. We achieved a loss of 1*10^-4. This is in meters meaning that the average precision of the model is less than 0.1mm off.
The jupyter notebooks for creating and training the models are attached.

Project subpart: Camera calibration

Since we are using Isaac Sim to generate our training dataset, it is crucial to know the exact parameters of the camera to produce images that closely match the real photos captured by the camera intended for inference.

By using the pinhole camera model we can find the relationship between the real picture and the one captured on the sensor.

$$f_{\text{pixels}} = \frac{\text{size of the object on the image (pixels)} \times \text{distance to the object (mm)}}{\text{physical size of the object (mm)}}$$

We 3D printed a chess board, put it at a distance of 20cm and took multiple pictures. Knowing that each chessboard square measures 2 cm, we determined the physical size of the object, the size of the object on the image was found by applying a grid on the picture and knowing each picture is 240x240 pixel we could count how many pixels does the edge of the chess board span over. It was found to be 32 pixels per square.

$$f_{\text{pixels}} = \frac{32 \times 200}{20} = 320\,\text{pixels}$$

Now to find the focal length in millimeters we need to multiply the focal length in pixels by the size of a pixel which for our camera is 2.2 µm.

$$f_{\text{mm}} = 320 \times 0.0022 = 3.52\,\text{mm}$$

Such calculations were done on the other pictures and the calculated focal length was averaged from the different calculations.

In conclusion, the calculated focal length of 3.5 mm ensures accurate camera calibration, enabling the generation of simulated images that closely match real-world conditions for improved inference reliability.

Project subpart: Inference on ESP32

Idea for running on ESP32

The model was specifically designed so that it would fit onto an ESP32-S3 chip used in the ESP32-S3-EYE board we had for this course. The idea is that the ESP would process one image at a time. This means that we would have one model for the convolutional layers and one model for the dense layers. Then we would first run a single image through the convolutional layers. Save this result. Then we would run the second image through the convolutional layers. Save that result and run both those results through the dense layers. Then we would delete the first result from the first image and run a third image through the convolutional layers. Then the result from the second and the third image would be run through the dense layers and the system would continue like this.

Converting model and using on ESP32

The first step was to split the model in TensorFlow so that the convolutional layers and the dense layers were separate. Then we converted both the convolutional layers as one model and the dense layers as another model into the .onnx format. These two models were then quantized using the ESP-DL quantization tool. After quantization both models can now be used on the esp. Now we first initialize the convolutional model, then we run the first image through the convolutional model and save the result. after this we run the second image through the model and combine both results into 1 matrix. We then initialize our regression head (the dense layers model) and run the matrix through that.

Results from ESP32

We only ran the model with static images saved into the esp's memory from the pc. We also had the issue that we could not fit 2 images into the ram of the esp. This would not be an

issue in the final version since we would only process one image at a time anyway but since we used images already taken from the pc we could only fit one image meaning that we ran the same image through the model twice. This makes the result hard to evaluate but from previous experience with other models the quantization does not really affect the accuracy of the model. The time however was very fast:

```
Conv::forward: 16262 us
Conv_2::forward: 15611 us
Reg::forward: 13473 us
0.3628m
```

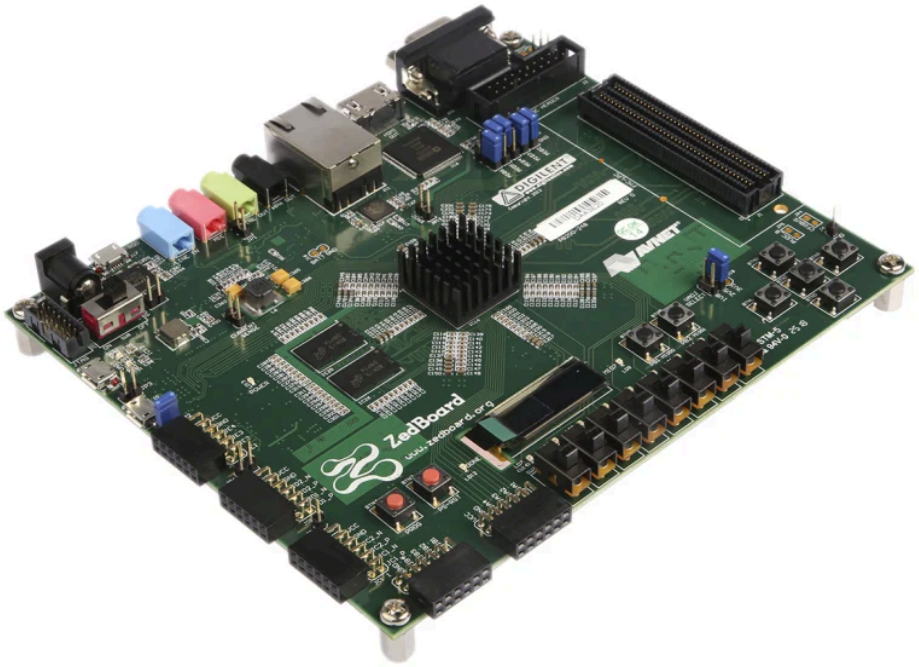
as seen in the screenshot one convolutional pass and one regression pass together would only take around 30ms which would allow for a framerate of over 30 fps and it also almost matches the time of inference on cpu on the pc in tensorflow(25 ms):

```
tf.Tensor(0.10347651, shape=(), dtype=float32)
0.025912284851074222
0.11788339969091588
```

From these findings it seems that with more time a full setup could be created where the ESP32 accurately would be able to estimate its own movement through 3D space only using the single camera it has.

The full project for the ESP32 is attached.

Project subpart: Inference on FPGA



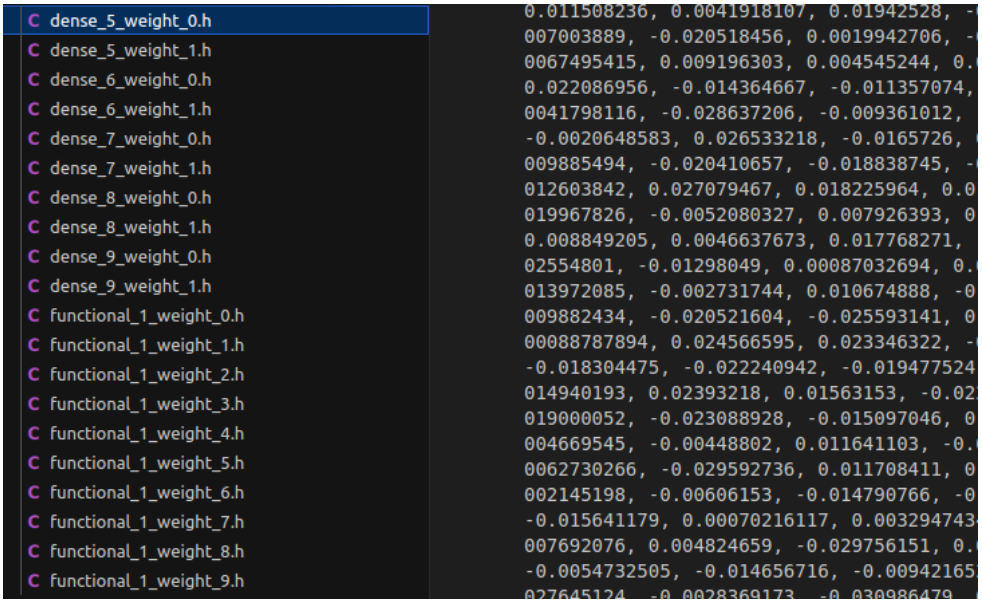
Extracting weights from trained network:

The weights from the trained network was saved in cpp header files using a python script:

Here the model is being loaded:

```python
import tensorflow as tf
import numpy as np
import os

# Load the model (update the file path to your model)
model = tf.keras.models.load_model("dist_from_2_img_snall_head.keras")

model.summary()
```

The layers are then separated and the associated weights are saved in cpp header files as arrays:



Converting network to cpp:

While tensorflow has a cpp api only the lite version is compatible with the ZedBoard's processor. Its even harder to find libraries that can be imported into Vitis HLS, which is the application that converts cpp code to VHDL that can be implemented on the FPGA.
With these considerations it was chosen to only use libraries that could be used for the code running on the processor:

The network has 4 different types of layers: Avg_pooling, Max_pooling, Convolution, and Dense, including two activation functions relu and sigmoid.
It's also necessary to normalise the image's passed to the neural network. All this was written into cpp functions. And can be viewed in the network.cpp file.

Memory was allocated to each layer:
And the combined network was constructed.

```cpp
int main() {
    for (int i = 0; i < 2; ++i) {
        if (!im_d) {
        }


        // Resize the image to 240x240 while keeping all 3 channels
        stbir_resize_uint8_linear(im_d, im_w, im_h, 0, im_d_r, img_sides, img_sides, 0, (stbir_pixel_layout)3);
        normalizeImage(im_d_r, img_sides, img_sides, 3, normalized_image);

        int poolSize = 3;
        int stride = 3;
        int pooledHeight = (img_sides - poolSize) / stride + 1;
        int pooledWidth = (img_sides - poolSize) / stride + 1;
        float* pooled_output = new float[pooledHeight * pooledWidth * 3];
        averagePool2D(normalized_image, img_sides, img_sides, 3, poolSize, stride, avg_pool1_output);
        conv2D(avg_pool1_output, conv1_weight, conv1_bias, avg_pool1_output_h, avg_pool1_output_w, color_channels, conv1_kernel, conv1_filters, conv1_stride, conv1_output);
        maxPool2D(conv1_output, conv1_output_h, conv1_output_w, conv1_filters, max_pool1_kernel, max_pool1_stride, max_pool1_output);
        conv2D(max_pool1_output, conv2_weight, conv2_bias, max_pool1_output_h, max_pool1_output_w, conv1_filters, conv2_kernel, conv2_filters, conv2_stride, conv2_output, "same");
        maxPool2D(conv2_output, conv2_output_h, conv2_output_w, conv2_filters, max_pool2_kernel, max_pool2_stride, max_pool2_output);
        conv2D(max_pool2_output, conv3_weight, conv3_bias, max_pool2_output_h, max_pool2_output_w, conv2_filters, conv3_kernel, conv3_filters, conv3_stride, conv3_output, "same");
        conv2D(max_pool2_output, conv4_weight, conv4_bias, max_pool2_output_h, max_pool2_output_w, conv2_filters, conv3_kernel, conv3_filters, conv3_stride, conv3_output, "same");
        conv2D(max_pool2_output, conv5_weight, conv5_bias, max_pool2_output_h, max_pool2_output_w, conv2_filters, conv3_kernel, conv3_filters, conv3_stride, conv3_output, "same");
        maxPool2D(conv5_output, max_pool2_output_h, max_pool2_output_w, conv3_filters, max_pool3_kernel, max_pool3_stride, max_pool3_output);

        std::memcpy(concatenated_output, last_max_pool3_output, flatten_size * sizeof(float)); // Copy last
        std::memcpy(concatenated_output + flatten_size, max_pool3_output, flatten_size * sizeof(float)); // Copy current


        if( i >0 )
        {
            denseLayer(concatenated_output, dense1_weight, dense1_bias, concatenated_size, 16, dense1_output);
            relu(dense1_output, 16);

            // Dense Layer 2
            denseLayer(dense1_output, dense2_weight, dense2_bias, 16, 16, dense2_output);
            relu(dense2_output, 16);

            // Dense Layer 3
            denseLayer(dense2_output, dense3_weight, dense3_bias, 16, 16, dense3_output);
            relu(dense3_output, 16);

            // Dense Layer 4
            denseLayer(dense3_output, dense4_weight, dense4_bias, 16, 16, dense4_output);
            relu(dense4_output, 16);

            // Dense Layer 5
            denseLayer(dense4_output, dense5_weight, dense5_bias, 16, 1, dense5_output);
            sigmoid(dense5_output, 1);

            // Print final output (for verification)
            std::cout << "Final Output: " << dense5_output[0] << std::endl;
        }
         stbi_image_free(im_d);    // Free original image data

         std::memcpy(last_max_pool3_output, max_pool3_output, flatten_size * sizeof(float));
    }
```

While working on this, the realization of the of the size of weights that need to be stored and processed on the FPGA and its limitations.
Especially its limited BRAM of 4.9 Mb (mega bits), this converts to 615000 bytes.(https://docs.amd.com/v/u/en-US/ds190-Zynq-7000-Overview)

XILINX

Zynq-7000 SoC Data Sheet: Overview

Table 1: **Zynq-7000 and Zynq-7000S SoCs *(Cont'd)***

	Device Name	Z-7007S	Z-7012S	Z-7014S	Z-7010	Z-7015	Z-7020	Z-7030	Z-7035	Z-7045	Z-7100
	Part Number	XC7Z007S	XC7Z012S	XC7Z014S	XC7Z010	XC7Z015	XC7Z020	XC7Z030	XC7Z035	XC7Z045	XC7Z100
Programmable Logic	Xilinx 7 Series Programmable Logic Equivalent	Artix®-7 FPGA	Artix-7 FPGA	Artix-7 FPGA	Artix-7 FPGA	Artix-7 FPGA	Artix-7 FPGA	Kintex®-7 FPGA	Kintex-7 FPGA	Kintex-7 FPGA	Kintex-7 FPGA
	Programmable Logic Cells	23K	55K	65K	28K	74K	85K	125K	275K	350K	444K
	Look-Up Tables (LUTs)	14,400	34,400	40,600	17,600	46,200	53,200	78,600	171,900	218,600	277,400
	Flip-Flops	28,800	68,800	81,200	35,200	92,400	106,400	157,200	343,800	437,200	554,800
	Block RAM (# 36 Kb Blocks)	1.8 Mb (50)	2.5 Mb (72)	3.8 Mb (107)	2.1 Mb (60)	3.3 Mb (95)	4.9 Mb (140)	9.3 Mb (265)	17.6 Mb (500)	19.2 Mb (545)	26.5 Mb (755)
	DSP Slices (18x25 MACCs)	66	120	170	80	160	220	400	900	900	2,020
	Peak DSP Performance (Symmetric FIR)	73 GMACs	131 GMACs	187 GMACs	100 GMACs	200 GMACs	276 GMACs	593 GMACs	1,334 GMACs	1,334 GMACs	2,622 GMACs
	PCI Express (Root Complex or Endpoint)(3)		Gen2 x4			Gen2 x4		Gen2 x4	Gen2 x8	Gen2 x8	Gen2 x8
	Analog Mixed Signal (AMS) / XADC	2x 12 bit, MSPS ADCs with up to 17 Differential Inputs									
	Security(2)	AES and SHA 256b for Boot Code and Programmable Logic Configuration, Decryption, and Authentication									

Notes:

1. Restrictions apply for CLG225 package. Refer to the UG585, *Zynq-7000 SoC Technical Reference Manual* (TRM) for details.
2. Security is shared by the Processing System and the Programmable Logic.
3. Refer to PG054, *7 Series FPGAs Integrated Block for PCI Express* for PCI Express support in specific devices.

Makes it impossible to load all the weights of even some of the single layers. A better approach, would be to load the weights into the main shared memory and read from that part of the memory when they are needed. This is different to the approach we learnt in the course Hardware Software Codesing, where we are sending data back and forward using an axi communication block and load data into a buffer.
Because of these limitations the decision to only move either the avg_pooling layer or the max_pooling layer to the FPGA was made.

```cpp
// average pooling has been run on picture
void averagePool2D(const float* input, int height, int width, int channels, int poolSize, int stride, float* output) {
    int outputHeight = (height - poolSize) / stride + 1;
    int outputWidth = (width - poolSize) / stride + 1;

    for (int c = 0; c < channels; ++c) {
        for (int i = 0; i < outputHeight; ++i) {
            for (int j = 0; j < outputWidth; ++j) {
                float sum = 0.0f;
                for (int m = 0; m < poolSize; ++m) {
                    for (int n = 0; n < poolSize; ++n) {
                        int row = i * stride + m;
                        int col = j * stride + n;
                        sum += input[(row * width + col) * channels + c];
                    }
                }
                output[(i * outputWidth + j) * channels + c] = sum / (poolSize * poolSize);
            }
        }
    }
}
```

Notice that this function is not yet vectorized.

Vitis HLS implementation:

The image that needs to be processed by the avg_pooling is a 3 channel RGB image.

In other words 240*240*3 bytes. This needs to be normalized and converted to float datatypes both for the avg filter but also if we were to implement the rest of the neural network. As a result loading and converting the image to float will already be more than 240*240*3*4 = 691200 bytes which is more BRAM than there is available. So in the implementation one channel is being sent and processed at a time.

Preparation:

```
1  #include <hls_stream.h>
2  #include <ap_axi_sdata.h>
3  #include <ap_int.h>
4  #include <math.h>
5
6  #define im_sides 240          // Input image
7  #define color_channels 1      // Number of c
8  #define pool_size 3           // Pool size f
9  #define stride 3              // Stride for
10
11 typedef ap_axis<32, 0, 0, 0> stream_data_t;
```

We include header files which are necessary for the communication aswell as processing logic. We then define some constants and typedef for the axi communication.

Loop 1: Receiving data

```
void hardware_func(hls::stream<stream_data_t>& in_stream, hls::stream<stream_data_t>& out_stream) {
#pragma HLS INTERFACE mode=axis port= in_stream, out_stream

//#pragma HLS INTERFACE axis port=in_stream
//#pragma HLS INTERFACE axis port=out_stream
//#pragma HLS INTERFACE s_axilite port=return bundle=CTRL

    const int output_sides = (im_sides - pool_size) / stride + 1;
    const int output_size = output_sides * output_sides;

    // Buffers for storing input and output data
    float input_buffer[im_sides * im_sides];
    unsigned char output_buffer[output_size];

    for(unsigned int i = 0; i < ((im_sides * im_sides)/4); i++)
    {
        //pragma to ensure that this loop is threated as a seperate loop
#pragma HLS PIPELINE II=1
        stream_data_t temp = in_stream.read();
        //split recieved int into 4 chars
        for(unsigned int j = 0; j < 4; j++)
        {
            unsigned char pixel_value = ((temp.data >> (j*8)) & 0xFF);
            input_buffer[i*4+j] = pixel_value / 255.0f;
        }
    }
```

Next we allocate memory for the input image and the compressed output image. The first loops reads the data from the channel splits the bytes which was interpreted as integers into their individual parts before normalising and storing them as floats.

Loop 2: Avg_pooling

```
    // Step 2: Perform average pooling for the entire image
    for (int idx = 0; idx < output_size; ++idx) {
#pragma HLS PIPELINE II=1
        int i = idx / output_sides;                    // Row index of pooled output
        int j = idx % output_sides;                    // Column index of pooled output

        float sum = 0.0f;
        for (int m = 0; m < pool_size; ++m) {

            for (int n = 0; n < pool_size; ++n) {

                int row = i * stride + m;
                int col = j * stride + n;
                sum += input_buffer[row * im_sides + col];
            }
        }

        float avg = sum / (pool_size * pool_size);
        output_buffer[idx] = (unsigned char)(avg * 255.0f);  // Convert back to unsigned char
    }
```

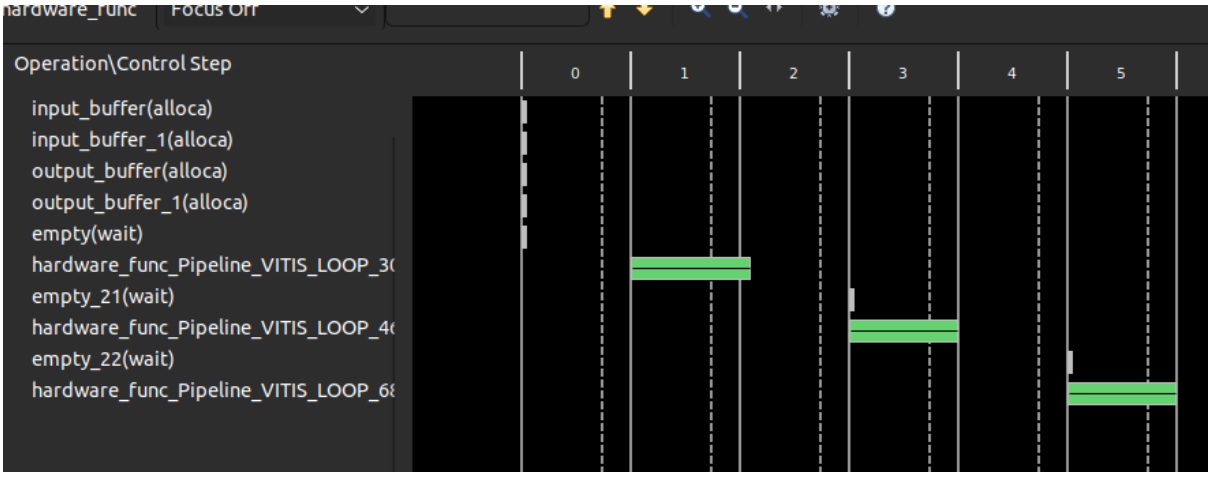
The algorithm for avg polling is applied and the result is converted back to an array of unsigned chars representing the compressed image that is send back to the processor.

Loop 3: sending data back

```
    for(unsigned int i = 0; i < (output_size / 4); i++)
    {
        //pragma to ensure that this loop is threated as a seperate loop
#pragma HLS PIPELINE II=1
        stream_data_t to_send;
        //combining 4 chars into one integer again
        to_send.data = output_buffer[i * 4] | (((int)output_buffer[i * 4 + 1]) << 8) | (((int)output_buffer[i * 4 + 2
        //keep all 4 bytes we have no transfers with less than four of the bytes used
        to_send.keep = 0b1111;
        //if on last iteration of the loop last will be 1
        to_send.last = (i == ((output_size/4) - 1)) ? 1 : 0;
        out_stream.write(to_send);
    }
```

The last loop sends the data back to the processor. We are combining all the chars into an int which we are sending. This matches the ap axi communication size fo 32 bits. We also give the encoding of 0b1111 to represent that all chars of integer is data that we should store.

Vitis HLS syntesis:

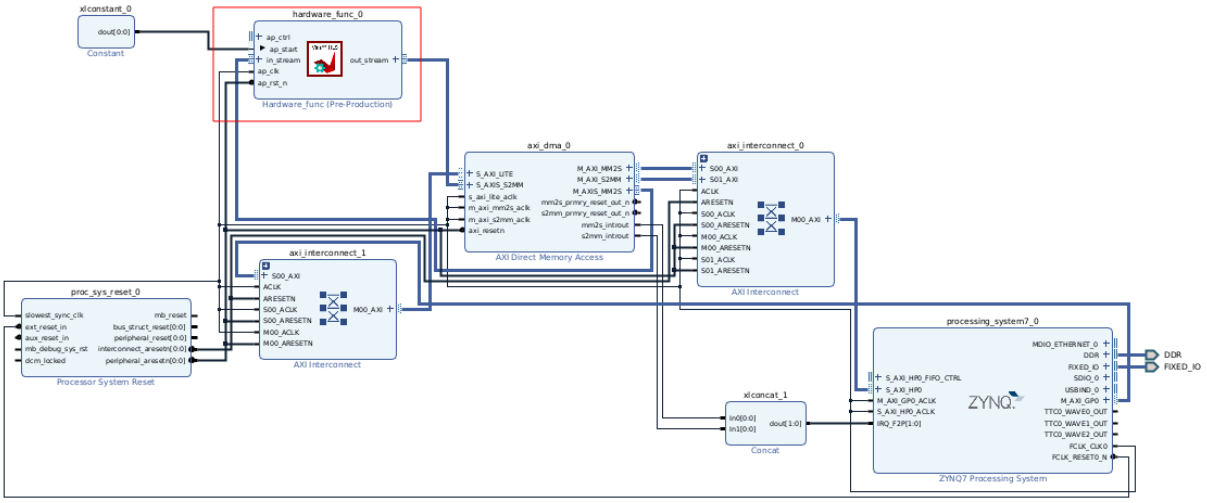


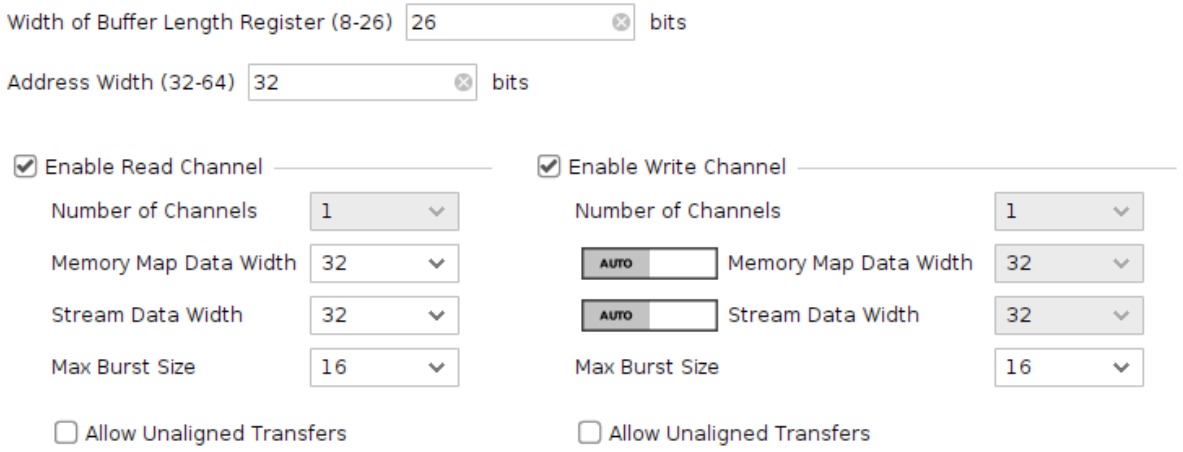
From the vitis synthesis we can see how the initialization and the 3 loops are all separated by clockcyles, so separating dependent part of the code to not run concurrently.

Slack	Latency(cycles)	Latency(ns)	Iteration Latency	Interval	Trip Count	Pipelined	BRAM	DSP	FF	LU
-	35304	3.530E5	-	35305	-	no	132	10	2904	391

We can also read some of definitions for the synthesised code. Here we are using 132 blocks of BRAM which is just below the limit of 140 available on the zedboard zynq 7020

Vivado Implementation:





We then import the

Processor:

The full code is name Sending_data.cpp and can be found under the shared files.

Include and defines:

```
#include "dma.hpp"
#define STB_IMAGE_IMPLEMENTATION
#include "stb_image.h"
#define STB_IMAGE_RESIZE_IMPLEMENTATION
#include "stb_image_resize2.h"
#define STB_IMAGE_WRITE_IMPLEMENTATION
#include "stb_image_write.h"

#define im_sides 240
#define im_size im_sides * im_sides * 3 // For RGB image (3 channels)
#define res_sides 80
#define res_size res_sides * res_sides // For one channel (grayscale)
```

The libraries and definition used on the processor.

Loading image:

```
int im_w, im_h, im_c;

// Load RGB image
unsigned char* im_d = stbi_load("rgb_240_0000.png", &im_w, &im_h, &im_c, STBI_rgb);
if (!im_d) {
    printf("no_image_loaded :(\n");
    return -1;
}

// Allocate memory for resized input and output
unsigned char* im_d_r = new unsigned char[im_size];
unsigned char* res_d = new unsigned char[res_size];
unsigned char* combined_result = new unsigned char[res_size * 3]; // Combined RGB result

// Resize input image to 240x240
stbir_resize_uint8_linear(im_d, im_w, im_h, 0, im_d_r, im_sides, im_sides, 0, (stbir_pixel_layout)3);
```

We load the image with all 3 channels and convert it to 240x240 resolution.

Dma initialize and send data:

```
DirectMemoryAccess* dma = new DirectMemoryAccess(0x40400000, 0x0e000000, 0x0f000000);

// Extract channels
unsigned char* red_channel = new unsigned char[im_sides * im_sides];
unsigned char* green_channel = new unsigned char[im_sides * im_sides];
unsigned char* blue_channel = new unsigned char[im_sides * im_sides];

for (int i = 0; i < im_sides * im_sides; ++i) {
    red_channel[i] = im_d_r[i * 3];       // Extract Red
    green_channel[i] = im_d_r[i * 3 + 1]; // Extract Green
    blue_channel[i] = im_d_r[i * 3 + 2];  // Extract Blue
}

// Process each channel
unsigned char* channels[] = {red_channel, green_channel, blue_channel};
const char* channel_names[] = {"red_channel.jpg", "green_channel.jpg", "blue_channel.jpg"};

for (int c = 0; c < 3; ++c) {
    printf("Processing channel %d\n", c);

    // Reset DMA
    dma->halt();
    dma->reset();

    // Send channel data to DMA
    for (unsigned int i = 0; i < im_sides * im_sides; ++i) {
        dma->writeSourceInteger(channels[c][i]);
    }

    // Debug: Hexdump source buffer
    dma->hexdumpSource(64);

    dma->setInterrupt(true, false, 0);
    dma->ready();
    dma->setDestinationAddress(0x0f000000);
    dma->setSourceAddress(0x0e000000);
    dma->setSourceLength(im_sides * im_sides);  // Source size for one channel
    dma->setDestinationLength(res_sides * res_sides);  // Destination size for one channel

    printf("waiting for MM2S\n");
    unsigned long int status;
    do {
        status = dma->getMM2SStatus();
        dma->dumpStatus(status);
    } while (!(status & (1 << 12)) && !(status & (1 << 1)));

    printf("waiting for S2MM\n");
    do {
        status = dma->getS2MMStatus();
        dma->dumpStatus(status);
    } while (!(status & (1 << 12)) && !(status & (1 << 1)));

    dma->resetCursor();
```

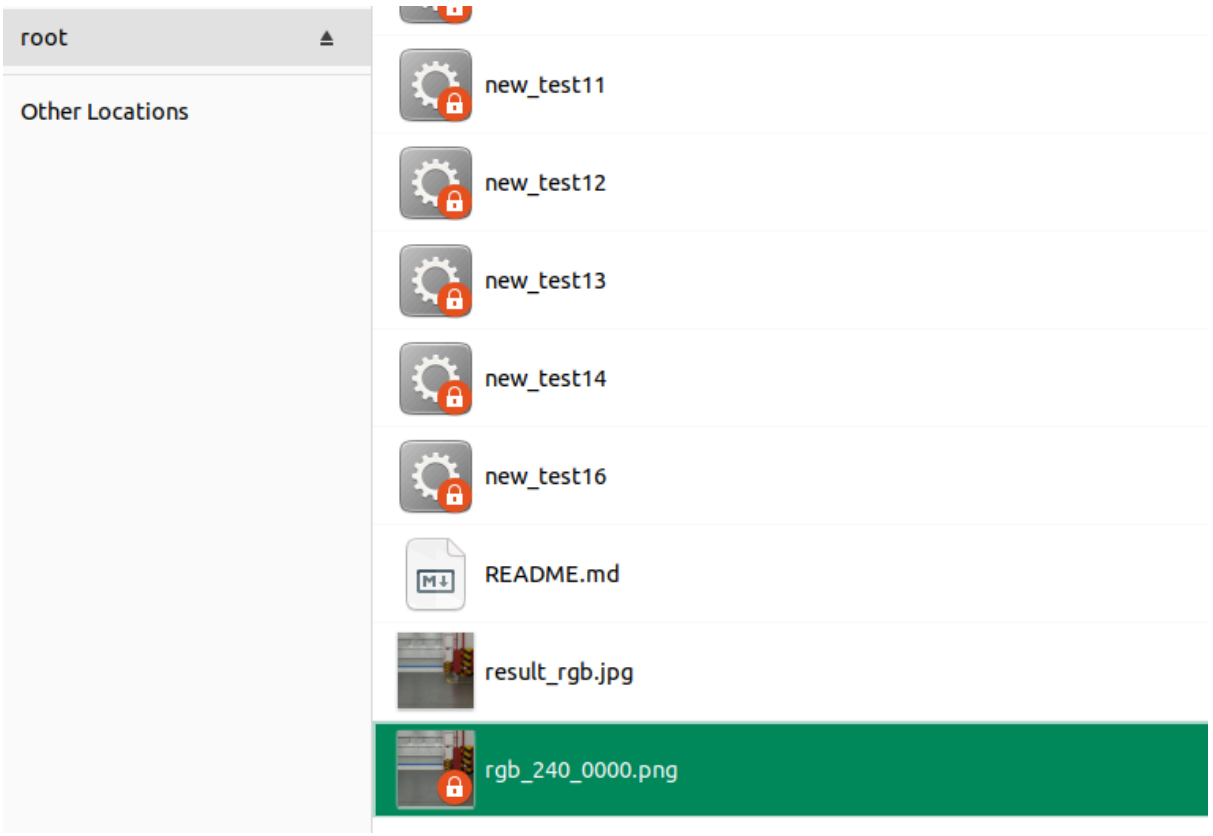
We initialize the dma and send the data for every channel independently through 3 iterations. In every iteration we wait for the FPGA to process the data and send it back.

Read response from FPGA:

```
    dma->resetCursor();

    // Debug: Hexdump destination buffer
    dma->hexdumpDestination(64);

    // Read back processed data
    for (unsigned int i = 0; i < res_sides * res_sides; ++i) {
        res_d[i] = dma->readDestinationByte();
    }

    // Save channel data as a grayscale image
    stbi_write_jpg(channel_names[c], res_sides, res_sides, 1, res_d, 100);

    // Combine channel into the final RGB image
    for (int i = 0; i < res_sides * res_sides; ++i) {
        combined_result[i * 3 + c] = res_d[i];
    }
}

// Save the combined RGB result
stbi_write_jpg("result_rgb.jpg", res_sides, res_sides, 3, combined_result, 100);
printf("Complete :)\n");

// Clean up
delete[] im_d_r;
delete[] res_d;
delete[] combined_result;
delete[] red_channel;
delete[] green_channel;
delete[] blue_channel;
delete dma;
stbi_image_free(im_d);

return 0;
```

We read the processed result and combine it into a full image over multiple iterations.

Results:



We can see from the result that the picture has been compressed using the avg filter (80x80x3) and it almost seems more pixelated and blurry. This is the first layer of the convolutional neural network. Video of one of the implementation is also included under files. The binary was updated since and is the newest new_test binary.

Future work and conclusion:

To improve on the project we would expand the model to work in other directions than just the camera forward direction. Also we would need to finish the code for the ESP so that it

would run the model on live images. In terms of the FPGA we would need to implement the rest of the layers and put everything together. It can however be concluded from our sub results that the system would work as intended once everything has been fully finished.