EIE 1st Year FPGA Project

Management Report

Group 18
Arsal Zaman, Horace Lee, Kimon Grigorakis, Moin Bukhari

3814 words

Table of Contents

Section 1 Understanding of Process and Tutorials

The Lab sessions were split into two parts:

Part 1: Introduction to Vivado HLS

Part 2: Introduction to PYNQ & Overlay Design

Both parts introduce you to the tools and techniques that you can implement in the final project.

1.1 Introduction to Vivado HLS(High-Level Synthesis)

DEEPER LOOK INTO HIGH-LEVEL SYNTHESIS

Vivado HLS is a tool that allows you to program in C++ and generate a hardware specification from it. It transforms high-level programming language(C++) into an RTL (Register Transfer Level) Implementation so that you can synthesise this with the FPGA(Field Programmable Gate Array) to provide a parallel architecture that can increase performance, reduce cost and power when compared to traditional processors.

1.1.1 Creating a High-Level Synthesis Project

There are 4 steps in producing a functioning HLS project:

1. Validation: Ensures the C++ algorithm is performing the correct operation and creating a testbench which confirms that the results are correct.
2. Synthesis: Produces a report on the performance estimates, utilization estimates and what ports and input/output(I/O) protocols were created.
3. RTL Verification: Ensures that the synthesis design is logically correct without any major timing issues.
4. IP Creation: Packages the design as an IP(Intellectual Property) block to use in the Vivado Design Suite

After writing a complete C++ program and testbench for a specific function, you implement the steps in order to ensure that you are ready to synthesise the hardware code with the FPGA. The steps ensure that any problem is identified before moving to the next stage, whilst also allowing you to see where you could optimize your program to use fewer resources and retain high performance. Hence this lab provides the basis on how to produce an HLS project for the Final Group Project.

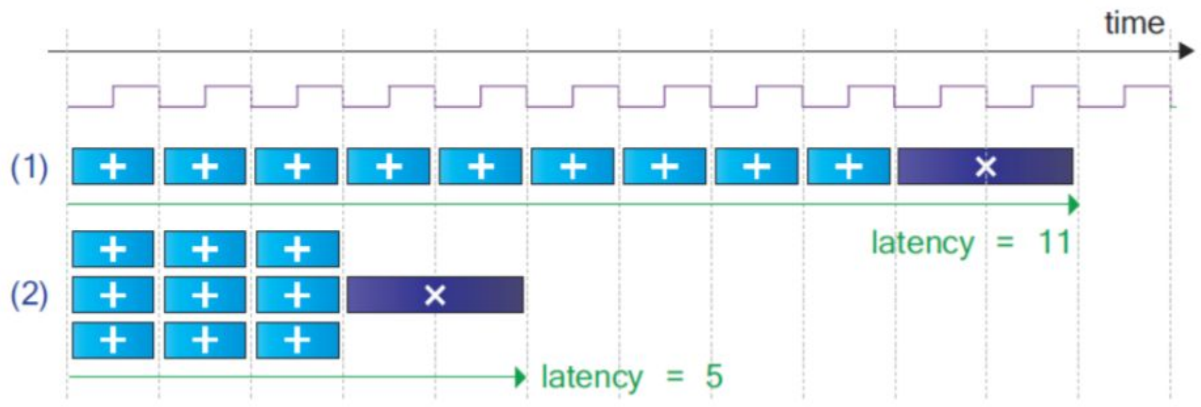
1.1.2 Using Solutions for Design Optimization

This Lab focused on creating solutions to optimize your implementation.

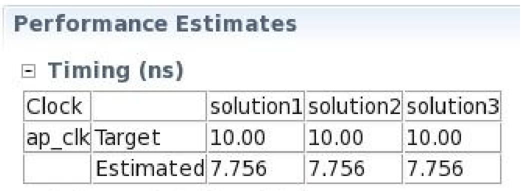
The 1st optimization performed must be to create the right I/O(input/output) protocols and ports. For example, if you do not explicitly specify the RAM access type, HLS might use a dual-port interface even though you only require single port access. This means you are utilizing more hardware unnecessarily.

You can also analyze the results for each solution to find areas to optimize and obtain the most optimal solution.

Loops can take many clock cycles to complete. To optimize this, loops can be unrolled, which means that you can carry out multiple iterations all in one cycle as long as each iteration is independent. As shown in Fig 1.1, this drastically reduces the latency(time taken for the whole loop to be executed).



[13] Fig 1.1 (1)Rolled Vs (2)Unrolled Loop

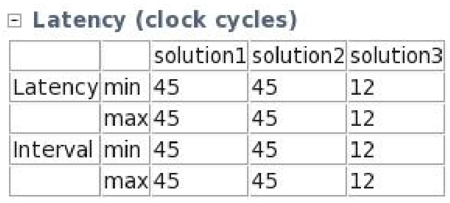


Performance Estimates

Timing (ns)

Clock		solution1	solution2	solution3
ap_clk	Target	10.00	10.00	10.00
	Estimated	7.756	7.756	7.756

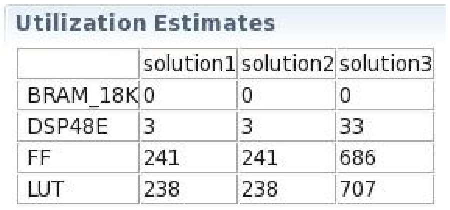
Fig 1.2 Performance Estimation For All 3 Solutions



Latency (clock cycles)

		solution1	solution2	solution3
Latency	min	45	45	12
	max	45	45	12
Interval	min	45	45	12
	max	45	45	12

Fig 1.3 Latency For All 3 Solutions



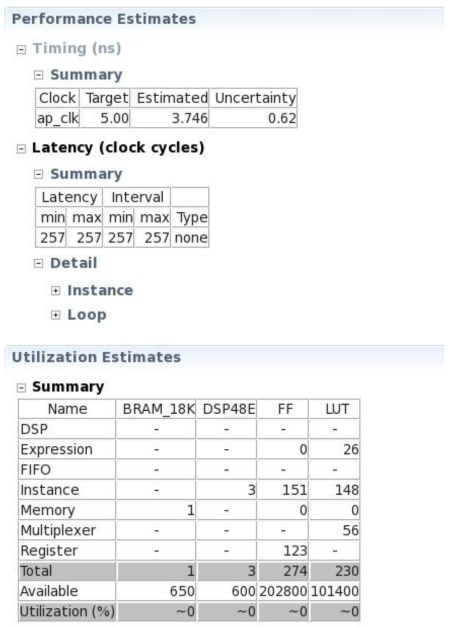
Utilization Estimates

	solution1	solution2	solution3
BRAM_18K	0	0	0
DSP48E	3	3	33
FF	241	241	686
LUT	238	238	707

Fig 1.4 Resource Utilization Estimation For All 3 Solutions

Figure 1.2, 1.3 and 1.4 show the synthesis report of 3 different solutions.
Solution 1 had no optimizations, solution 2 had a specified RAM-Access type, and solution 3 had unrolled the loop. Solution 3 is clearly the most optimal as it has the smallest latency however at a cost of higher resources utilization.
It seems Vivado HLS has automatically designated a 1 Port RAM-Access in solution 1 meaning that both solution 1 and 2 have exactly the same synthesis report.

The final optimization was the use of arbitrary precision types. This allows you to decide what size of hardware to implement in your design. For example, changing a data type that only needs to be 18-bit from int 32-bit can simplify the hardware needed to perform any calculations. As shown in Fig 1.5 and Fig 1.6, a fixed point design can reduce both latency and hardware utilization.



Performance Estimates

⊟ Timing (ns)

⊟ Summary

Clock	Target	Estimated	Uncertainty
ap_clk	5.00	3.746	0.62

⊟ Latency (clock cycles)

⊟ Summary

Latency		Interval		
min	max	min	max	Type
257	257	257	257	none

⊟ Detail

⊞ Instance

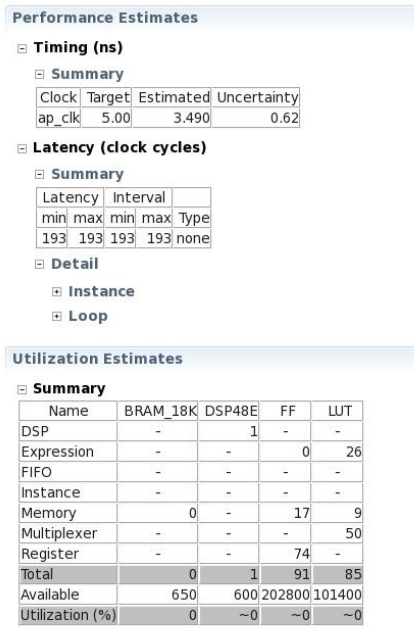
⊞ Loop

Utilization Estimates

⊟ Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	0	26
FIFO	-	-	-	-
Instance	-	3	151	148
Memory	1	-	0	0
Multiplexer	-	-	-	56
Register	-	-	123	-
Total	1	3	274	230
Available	650	600	202800	101400
Utilization (%)	~0	~0	~0	~0

Fig 1.5 Synthesis Report For Floating Point Design



Performance Estimates

⊟ Timing (ns)

⊟ Summary

Clock	Target	Estimated	Uncertainty
ap_clk	5.00	3.490	0.62

⊟ Latency (clock cycles)

⊟ Summary

Latency		Interval		
min	max	min	max	Type
193	193	193	193	none

⊟ Detail

⊞ Instance

⊞ Loop

Utilization Estimates

⊟ Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	1	-	-
Expression	-	-	0	26
FIFO	-	-	-	-
Instance	-	-	-	-
Memory	0	-	17	9
Multiplexer	-	-	-	50
Register	-	-	74	-
Total	0	1	91	85
Available	650	600	202800	101400
Utilization (%)	0	~0	~0	~0

Fig 1.6 Synthesis Report For Fixed Point Design

This tutorial shows the advantage of using bit accurate data-types to improving hardware implementation, Unrolling Loops to reduce latency and using the correct I/O protocols to get the best optimizations. Thus allowing you to create new solutions, add optimizations and compare the results of different solutions to select the best one.

1.1.3 Design a Dot-Product Module

The final part of the tutorials is designed in order to combine the techniques learned in the previous labs to create a dot product module which calculates the dot product of 2 vectors. The purpose of this lab was to show that after learning the basics of HLS you can now design any module for any function for your final project.

1.2 Introduction to PYNQ and Overlay Design

1.2.1 PYNQ Board

The PYNQ – Z1 board is a device that comprises of both PL (Programmable Logic), using FPGA logic fabric, and PS (Processing System) using a traditional CPU (Central Processing Unit). The FPGA allows you to design customized hardware, which can perform parallel computations generating high-performance systems with low power consumption [10].

It is worth mentioning that the PL logic circuits on the board can be handled through hardware libraries, called overlays, while programming the PS of the board and interfacing with the overlays is achieved through the use of an Interactive Python (IPython) kernel on the open-source Jupyter notebook infrastructure.

1.2.2 From C++ Code to Custom Hardware

Figure 1.7 summarizes all the steps required in order to generate customized hardware. To begin with, the very first step is to create your source code, written in C++, as well as a testbench, which will be used to verify the desired functionality of the source [11].

After verifying the functionality of the source code, the Vivado HLS (High-Level Synthesis) tool is used to convert the source code to an RTL (Register Transfer Level) HDL (Hardware Description Language) file. Exporting this file and loading into Vivado IDE enables you to perform physical/logic synthesis (i.e. placing and routing the RTL file into the PYNQ – Z1 board).

After the physical synthesis is complete and the block diagram of the custom hardware has been generated, two new files are created: the bitstream file, which will be used to program the PL part of the board, implementing the desired custom hardware, and the TCL file (i.e. tickle; the block diagram file).

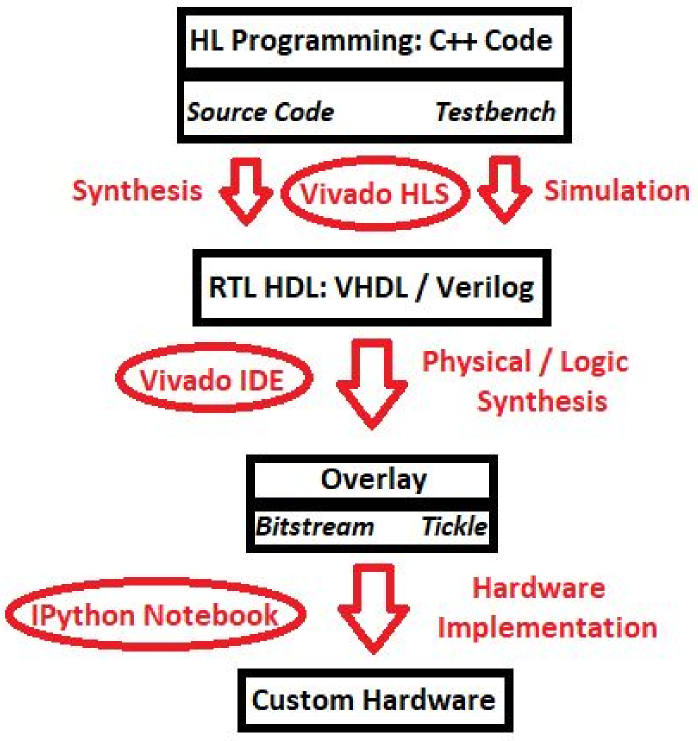


Fig 1.7 Overview of steps required to build custom hardware

Both files are then loaded on the PYNQ-Z1 board as an overlay and the custom hardware is configured on the PL part of the board, while the appropriate IPython kernel interface is established between the PL and PS, enabling you to read/write both control signals and memory locations. It is worth noticing that adjusting the control signals at runtime is necessary to trigger fundamental control operations like starting or stopping an operation on the PL part of the board.

1.2.3 Project Aim

The aim of the project is to create a real-time video processing hardware system by employing the capabilities of the PL part of the PYNQ-Z1 board. To be more specific, the PYNQ-Z1 board will receive video input from its HDMI input port, and after processing each frame in a pixel-wise manner within the FPGA logic fabric, it will output the processed frame to the HDMI output port. The way in which the video input will be processed will be presented in the Section 2.1 below, where are project idea is explored in detail.

1.2.4 Reference Design

The project implementation will be based on a reference design that implements vertical edge detection on the right half of the input image and is able to perform this detection in real time (i.e. many input frames per second). Edge detection has fundamental significance for our project and thus, it is essential to realize what it is and how it can be implemented.

To begin with, every image is stored in a pixel-wise manner, with each pixel represented by three 8-bit integer values, each one corresponding to the red, green, and blue (RGB) intensity values of the pixel. The “numerical nature” of the image allows us to perform various operations on it, including edge detection. In principle, the aim of the edge detection operation is to find the differences in intensity values between neighbouring pixels and output these differences on the output image. To do that, the grey level of each pixel is calculated from its RGB data and then used by the edge detection operation accordingly [12]. Specifically, to implement a basic vertical edge detector, the differences between the grey levels of horizontally neighboring pixels would have to be calculated and stored to the output image, while to implement a basic horizontal edge detector, the differences between the grey levels of vertically neighbouring pixels would have to be calculated and stored to the output image.

Section 2 Project

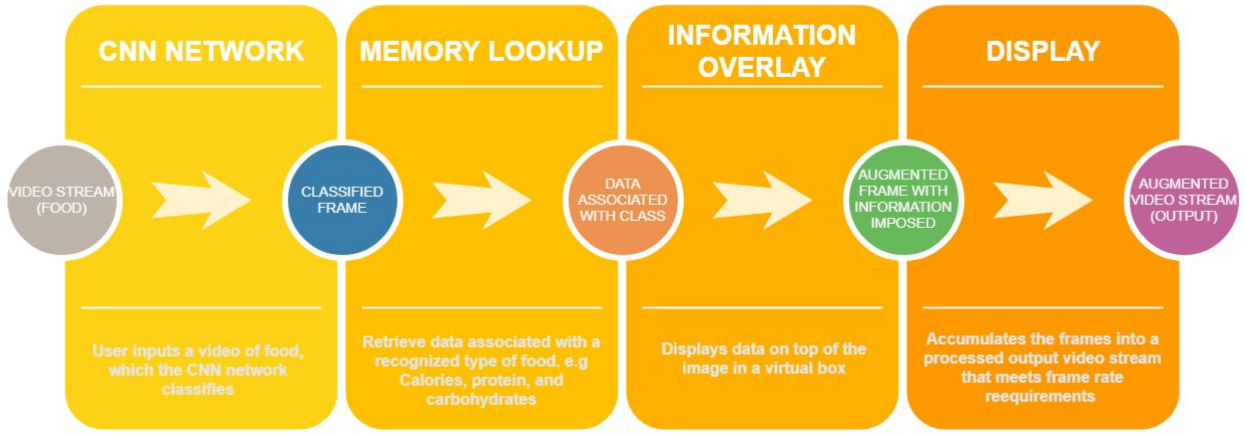
The overall specification given for this project is to produce a real-time (~25 fps), interactive image processing system using the PYNQ-Z1 board. In this section, we present two variations of our initial project idea which make use of the massively parallel processing capabilities of the PYNQ-Z1 board to perform hardware-accelerated image processing.

2.1 Description of the project

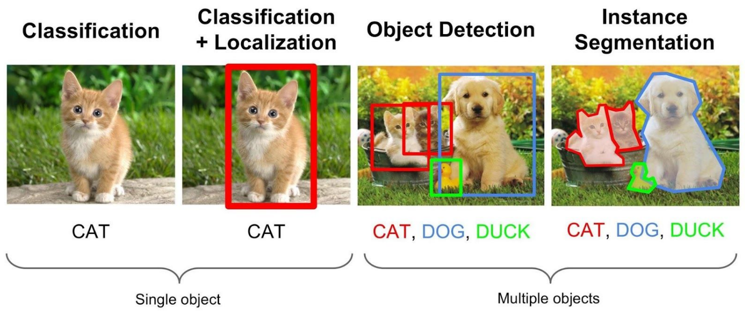
High-level description of the project idea

With increasing literacy, people are being more and more aware of the food they consume. Whether it is an average person trying to live a healthy lifestyle, an athlete who needs a certain threshold of calories to perform well, or simply a university student who wants to get fit, everybody needs a better way to keep track of their diet and be aware of the nutrients they are consuming.

Our idea is to create a deep learning based computer vision application that recognizes several commonly consumed food items. Users can simply hold items such as apples, mangoes, oranges, hamburgers, bread, etc. in front of the camera of our embedded device and it will display relevant nutritional information such as the typical amount of calories, fat, proteins, vitamins, cholesterol, etc. contained in the recognized food items, allowing users to keep track of their nutritional intake. The processing will be performed using a custom optimized implementation of the open source "MobileNetV2" convolutional neural network (CNN) architecture (more will be explained below), and depending on performance constraints we may decide to either perform object detection or just image classification. (Object detection involves identifying the locations of multiple objects in an image by displaying a bounding box around the detected object(s), whereas image classification involves simply recognizing whether an image contains a given object or not, which requires less computational power.)

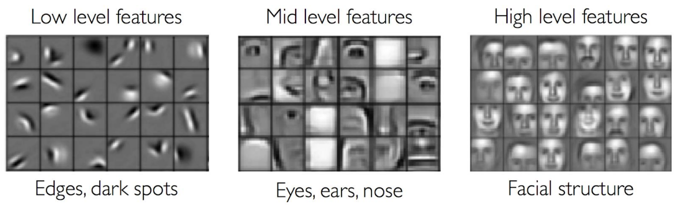

CNN NETWORK
MEMORY LOOKUP
INFORMATION OVERLAY
DISPLAY
VIDEO STREAM (FOOD)
CLASSIFIED FRAME
DATA ASSOCIATED WITH CLASS
AUGMENTED FRAME WITH INFORMATION IMPOSED
AUGMENTED VIDEO STREAM (OUTPUT)
User inputs a video of food, which the CNN network classifies
Retrieve data associated with a recognized type of food. e.g Calories, protein, and carbohydrates
Displays data on top of the image in a virtual box
Accumulates the frames into a processed output video stream that meets frame rate reequirements

Overview of CNNs



The convolutional neural network (CNN) is a powerful technique that has been successfully applied in various computer vision tasks such as image classification, object detection, and instance segmentation, in some cases surpassing human-level accuracy on a few benchmarks such as the ImageNet challenge [1,2].

By successively applying a ‘convolution’ operation layer by layer on an input image, CNNs are able to recognize complex features in the image, by first extracting low-level features such as edges and corners, and then combining information about these features to recognize more complex shapes and textures, and eventually recognizing full objects such as faces, animals, cars, etc [3].



In general CNNs have been more effective than traditional computer vision techniques in handling issues such as viewpoint variation (where an object’s appearance varies depending on viewpoint), scale variation, illumination, intra-class variation (where different objects within the same category can have very different appearances), occlusion (when parts of an object are blocked by another object or out of view in an image), and background clutter [3], and as such we believe CNNs would be a highly suitable technique to use in our project.

[14]

(Two images of road signs with different viewpoints, scales, illumination, etc.)

However, many of the popular CNN architectures require a large amount of processing, with larger models requiring billions of multiply-accumulate (MAC) operations. Running ‘inference’ (i.e. using a CNN to perform processing on a real-world image/video) with real-time performance on an embedded system (rather than on the cloud) remains a challenge. Although GPUs have been widely used to provide roughly an order of magnitude of acceleration compared to CPUs (due to their ability to perform more parallel computations), GPUs remain expensive and energy-intensive, making them less well suited to low-cost, low-power embedded applications. Currently, FPGAs (among other types of custom hardware) are being increasingly used for inference acceleration due to their massively parallel architectures and ability to be reconfigured to meet specific requirements [5].

Feasibility

In order to gauge the feasibility of running a CNN on the PYNQ board, we tested a demo notebook provided by Xilinx for the PYNQ board. The demo loaded a bitstream onto the FPGA to run a lightweight CNN (with only 6 convolutional, 3 max-pooling, and 3 fully

connected layers) to detect 42 different types of street signs from a popular image dataset known as the German Traffic Sign Recognition Benchmark (GTRSB) dataset. The demo achieved a ~3000 fps performance on our PYNQ-Z1 board [4], demonstrating that it is possible to at least run a basic CNN at a high frame rate, and this was mainly because it used a low number of layers as well as a new technique known as 'binarization' (where the weights of the network are represented with just 1-bit numbers rather than with 32-bit floating point numbers as used conventionally). Both techniques are used to trade off accuracy for lower latency.

In addition, we also experimented with another demo from Xilinx which runs 'Tincy YOLO', a modified version of the popular 'Tiny YOLO' object detection model. Due to this model being a more sophisticated CNN with more layers and filters, the demo performed at just 2.5 fps. However, the first and last layers of the CNN were not hardware-accelerated and thus they contributed to a significant overhead; excluding these two layers, we estimate that the network would have performed at ~10 fps. As with the previous demo, this demo also used binarized weights as well a reduced number of layers (the demo had fewer layers than the original 'Tiny YOLO' model).

Due to the high number of computations required for a CNN to achieve reasonable levels of accuracy, the main challenge this project is likely to face is our ability to meet the 25 fps frame rate requirement. Drawing from the above two examples, we will likely have to heavily rely on 'quantization' (i.e. using reduced-precision weights to increase computational speed) or even binarization, and we may have to use a low number of layers. On top of this, we must also effectively parallelize computations due to the sheer number of computations required to process a single image.

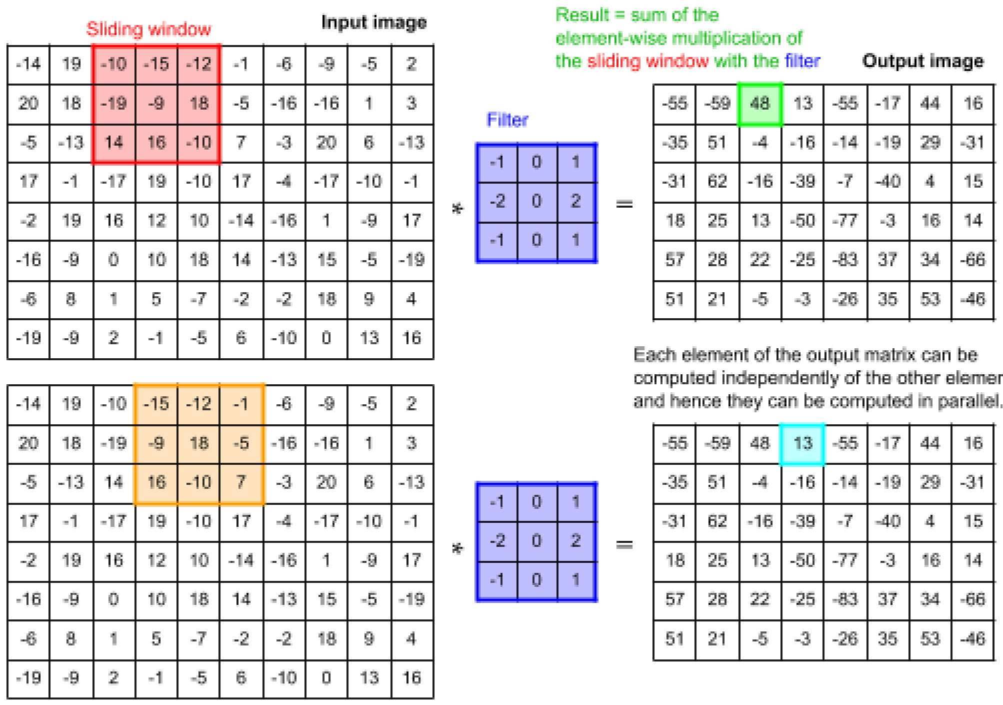
As mentioned earlier, we plan on implementing a version of the "MobileNetV2" CNN architecture which was designed for fast inference on embedded and mobile hardware. By itself, MobileNetV2 just performs image classification, but a variant of the model, "MobileNetV2+SSDLite", developed by the same authors, performs object detection by incorporating the SSDLite (Single Shot Detector Lite) architecture for object detection. Because our main goal is to ideally perform object detection, we will initially try to implement MobileNetV2+SSDLite on the FPGA, and fall back to just MobileNetV2 if we are unable to meet the 25 fps requirement, as the latter requires roughly 2.7x fewer computations to process a single image [6].

The MobileNetV2+SSDLite model has just 800 million MAC (multiply-accumulate) operations [6], which corresponds to roughly 1.6 GFLOPs assuming 1 MAC is performed in 2 operations. This makes the model roughly 2.8x less computationally expensive than the Tincy YOLO model we tested earlier, which uses 4.44 GFLOPs to process a single image [7]. Combined with various techniques described below, it should be feasible for us to achieve a near 25 fps performance on the PYNQ board, and if it proves to be too challenging, we can simply fall back to performing just image classification with the base MobileNetV2 model.

Techniques for accelerating CNN inference
Below are some techniques we plan on implementing, in order to ensure the real-time performance of our CNN:

1. Parallelizing convolutions
   The convolution operations in a CNN can be massively parallelized because each element of

the output image/matrix can be computed independently of the other elements (and hence they can be computed in parallel). For example, the diagram below illustrates a basic 2D convolution operation between an input image/matrix with a 3x3 filter (with padding=0, stride=1), and the green highlighted output element (row 1, column 3) can be computed independently of the cyan highlighted output element (row 1, column 4).


Sliding window
Input image
Result = sum of the element-wise multiplication of the sliding window with the filter
Output image
Filter
*
=
Each element of the output matrix can be computed independently of the other elements and hence they can be computed in parallel.

-14	19	-10	-15	-12	-1	-6	-9	-5	2
20	18	-19	-9	18	-5	-16	-16	1	3
-5	-13	14	16	-10	7	-3	20	6	-13
17	-1	-17	19	-10	17	-4	-17	-10	-1
-2	19	16	12	10	-14	-16	1	-9	17
-16	-9	0	10	18	14	-13	15	-5	-19
-6	8	1	5	-7	-2	-2	18	9	4
-19	-9	2	-1	-5	6	-10	0	13	16

-1	0	1
-2	0	2
-1	0	1

-55	-59	48	13	-55	-17	44	16
-35	51	-4	-16	-14	-19	29	-31
-31	62	-16	-39	-7	-40	4	15
18	25	13	-50	-77	-3	16	14
57	28	22	-25	-83	37	34	-66
51	21	-5	-3	-26	35	53	-46

2. Quantization or binarization - research has shown that using reduced precision arithmetic in CNNs can significantly speed up inference with minimal degradation in classification accuracy [8], as reduced precision numbers can be multiplied and added in fewer clock cycles, and fewer memory reads are required to retrieve the parameters of the network when performing convolutions and other operations.
3. Reducing input image resolution - instead of directly using a high-resolution video stream (e.g. 720p or 1080p), we will reduce the resolution of the video stream before passing the frames into our CNN for processing, as a lower resolution image requires less processing from the CNN.
4. Reducing number of layers - in case we are unable to meet the 25 fps requirement after attempting the other techniques, we could reduce the number of layers in our CNN (as a CNN uses various 'layers' to process an image, and by cutting down on some layers, our system will perform fewer computations) at the expense of reduced accuracy. However, removing certain layers will likely require us to re-train our network so this technique should probably be used as a last resort.

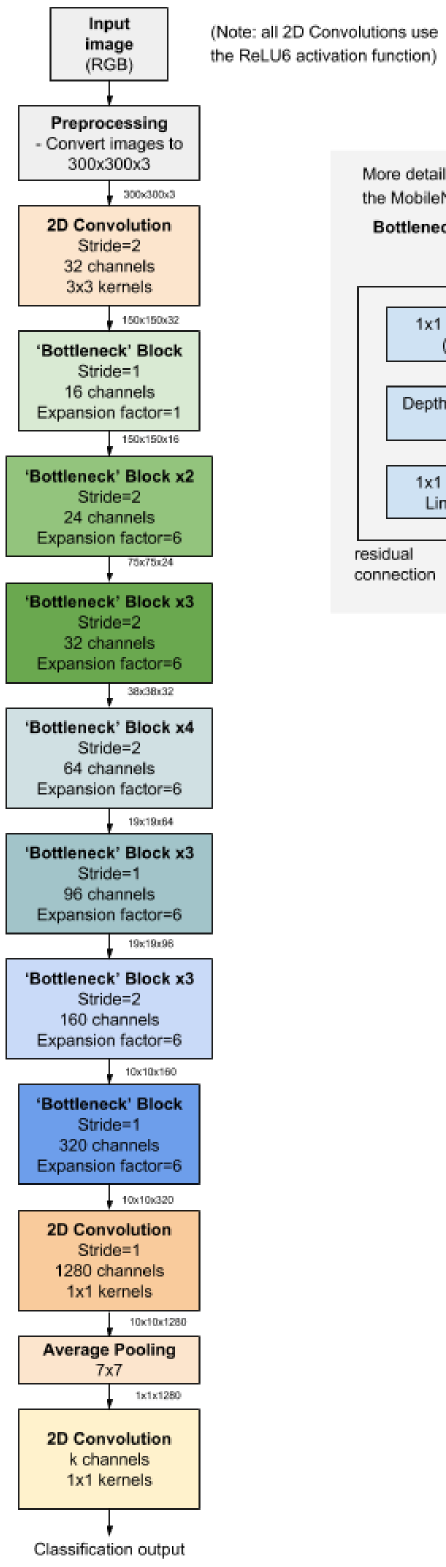
Training
To train our CNN to recognize various food items, we will use images from the Open Images Dataset, a large annotated dataset published by Google. Below is a selection of ‘classes’ (categories) we may decide to use from the dataset [9]:

Class	Number of images with bounding box labels
Apple	1529
Banana	511
Bagel	838
Doughnut	968
Croissant	512
Broccoli	472
Orange	2503
Hot dog	471
Hamburger	1758
Bread	2362
Pizza	1563

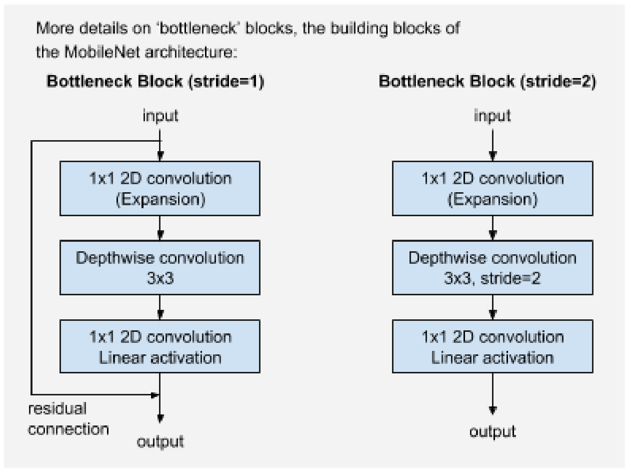
We will perform training using an open-source library called the Tensorflow Object Detection API (which one of our team members has prior experience using), which will allow us to train a quantized version of the MobileNetV2+SSDLite model on our custom dataset.

2.2 Schematic / diagram of the system

After researching into the MobileNetV2 CNN architecture, we have created the following diagram that illustrates the various blocks of the CNN that we will need to implement:



Input image (RGB)
(Note: all 2D Convolutions use the ReLU6 activation function)
Preprocessing - Convert images to 300x300x3
300x300x3
2D Convolution Stride=2 32 channels 3x3 kernels
150x150x32
'Bottleneck' Block Stride=1 16 channels Expansion factor=1
150x150x16
'Bottleneck' Block x2 Stride=2 24 channels Expansion factor=6
75x75x24
'Bottleneck' Block x3 Stride=2 32 channels Expansion factor=6
38x38x32
'Bottleneck' Block x4 Stride=2 64 channels Expansion factor=6
19x19x64
'Bottleneck' Block x3 Stride=1 96 channels Expansion factor=6
19x19x96
'Bottleneck' Block x3 Stride=2 160 channels Expansion factor=6
10x10x160
'Bottleneck' Block Stride=1 320 channels Expansion factor=6
10x10x320
2D Convolution Stride=1 1280 channels 1x1 kernels
10x10x1280
Average Pooling 7x7
1x1x1280
2D Convolution k channels 1x1 kernels
Classification output



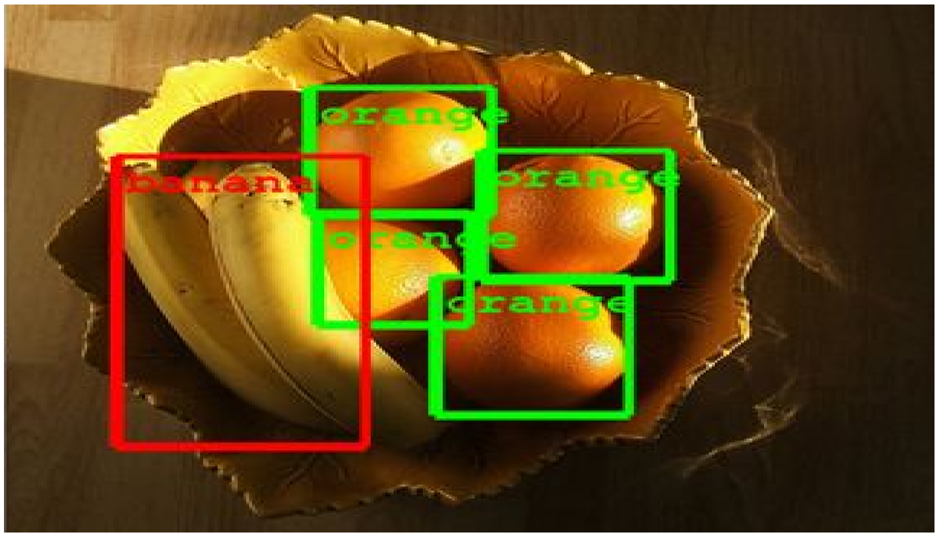
More details on 'bottleneck' blocks, the building blocks of the MobileNet architecture:
Bottleneck Block (stride=1)
input
1x1 2D convolution (Expansion)
Depthwise convolution 3x3
1x1 2D convolution Linear activation
residual connection
output
Bottleneck Block (stride=2)
input
1x1 2D convolution (Expansion)
Depthwise convolution 3x3, stride=2
1x1 2D convolution Linear activation
output

[6]

2.3 Expected outcomes

1. **Framerate**:
    a. The most crucial restriction on the scope of our project is that our program has to meet the 25 fps frame rate. Due to the significant amount of computations required for a CNN to process a single frame, we expect to employ the techniques outlined earlier (e.g. ensuring our processing is highly parallelized, using quantization, etc.) and spend a significant amount of time ensuring our code is highly optimized in order to meet the frame rate requirement.
    b. Since our CNN will be processing the live video feed frame by frame, the latency of our system (i.e. the time between getting an input image, processing it, and displaying it with some overlaid graphics) should be no more than 40 milliseconds.

2. **Neural Network:**
    a. Although we will likely be implementing techniques to speed up our CNN at the expense of lower accuracy (e.g. quantization, reduced input image resolution, reduced number of layers, etc.), we expect the overall classification accuracy of our system to be at least 60% at a minimum in order for it to be considered usable. In particular, we do not want our system to confuse one class for another class too often, such as confusing an orange for an apple.
    b. If we decide to implement object detection instead of image classification, we expect it to be able to recognize multiple objects in the same image (with accurate bounding boxes), such as in the example below where the multiple objects in the image have been recognized separately. This will allow the user to track and view nutritional information for multiple food items at once, instead of doing so one at a time.

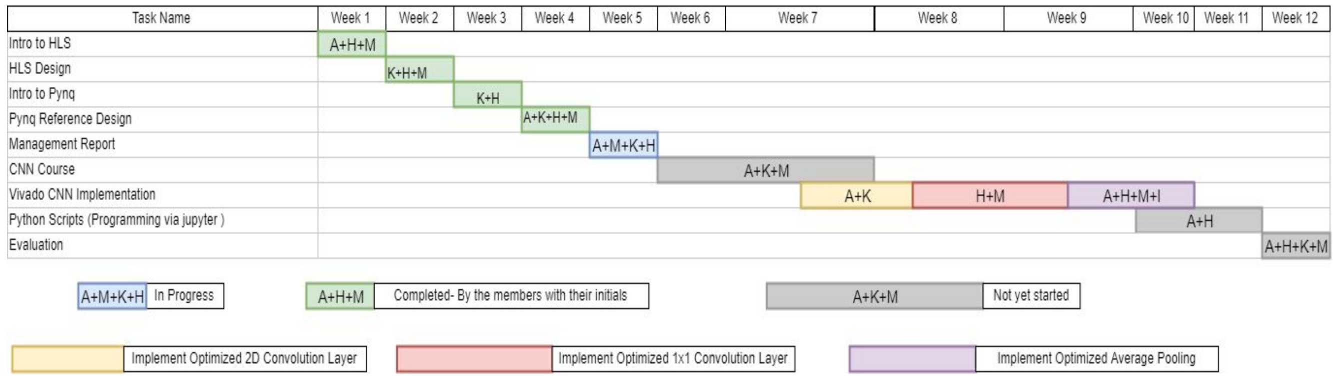


[15]

2.4 Project planning

1. **CNN online course**:
    a. While some of us have prior knowledge that will aid us throughout this project, we have identified some courses online that are tailored to what we want to create. So the first step for us to begin going through those courses, while the team members who are familiar will begin work.
2. **Dataset**:
    a. We will need to create a dataset with around 1000 images per class in order to train a high-accuracy network. These images (along with their annotations) can be easily downloaded from the Open Images Dataset.
3. **Implementation**:
    a. Implement and optimize the MobileNetV2 CNN architecture in C++ within the Vivado IDE. Below are some sub-tasks for implementing this, based on the different layers in the MobileNetV2 architecture:
        i. Implement and optimize code for performing 2D convolution
        ii. Implement and optimize code for performing 1x1 convolution
        iii. Implement and optimize code for performing depthwise convolution
        iv. Implement and optimize code for performing average pooling
        v. Combine and optimize the above layers to form the CNN
    b. Export the final logic design of this implementation and load the bitstream using Python to the PYNQ board.
    c. Write python code for miscellaneous bits like accessing video bitstream from HDMI ports.
    d. Run and analyse the performance of the board
    e. Adjust/debug accordingly
4. **Testing**:
    a. First and foremost we test the frame rate out using a video stream input.
    b. Then we will test for accuracy i.e how closely can our system recognise the food.
        i. We do this by trying varying input parameters, i.e what if we show a type of food, which we know our program will not recognise because we have not trained that particular class.
        ii. Does it give no input? Does it approximate? How accurate are its approximations?
    c. Evaluate Hardware resource utilization, and check if we can further improve it.
        i. Are there are components that do not significantly improve the accuracy but utilise huge amount of resources?
        ii. Is there more room for parallelization?
        iii. Is latency and throughput optimised? That is is any part of hardware waiting for some other part to get done and hence wasting time?

5. **Distribution of time and work** :
    a. Following Gantt Chart explains how we have distributed the workload.
    b. The Letters on the boxes are the initials of the group members responsible for completing that particular task.
    c. Vivado HLS CNN Implementation is subdivided into three subtasks, which all of us will do in pairs. We will then choose whichever one is most optimal. This way all of us can tap into our creative potential and maximise efficiency- as opposed to creating just one implementation all together.



References

[1]D. Gershgorn, "The data that transformed AI research—and possibly the world", *Quartz*, 2017. [Online]. Available: https://qz.com/1034972/the-data-that-changed-the-direction-of-ai-research-and-possibly-the-world/. [Accessed: 16- Mar- 2019]

[2]A. Ouaknine, "Review of Deep Learning Algorithms for Object Detection", *Medium*, 2018. [Online]. Available: https://medium.com/zylapp/review-of-deep-learning-algorithms-for-object-detection-c1f3d437b852. [Accessed: 16- Mar- 2019]

[3]A. Soleimany, *Deep Learning for Computer Vision*. MIT, 2019, pp. 11-12 [Online]. Available: http://introtodeeplearning.com/materials/2019_6S191_L3.pdf. [Accessed: 16- Mar- 2019]

[4]"BNN on Pynq", *GitHub*, 2019. [Online]. Available: https://github.com/Xilinx/BNN-PYNQ/blob/master/notebooks/CNV-BNN_Road-Signs.ipynb. [Accessed: 16- Mar- 2019]

[5]K. Abdelouahab, M. Pelcat, J. Serot and F. Berry, *Accelerating CNN inference on FPGAs: A Survey*. 2018 [Online]. Available: https://arxiv.org/abs/1806.01683. [Accessed: 16- Mar- 2019]

[6]M. Sandler, A. Howard, M. Zhu, A. Zhmoginov and L. Chen, *MobileNetV2: Inverted Residuals and Linear Bottlenecks*. Google, 2018 [Online]. Available: https://arxiv.org/pdf/1801.04381.pdf. [Accessed: 16- Mar- 2019]

[7]T. Preußer, G. Gambardella, N. Fraser and M. Blott, *Inference of Quantized Neural Networks on Heterogeneous All-Programmable Devices*. Xilinx, 2018 [Online]. Available: https://arxiv.org/pdf/1806.08085.pdf. [Accessed: 16- Mar- 2019]

[8]B. Jacob, S. Kligys, B. Chen, M. Zhu, M. Tang, A. Howard, H. Adam and D. Kalenichenko, *Quantization and Training of Neural Networks for Efficient Integer-Arithmetic-Only Inference*. Google, 2017 [Online]. Available: https://arxiv.org/pdf/1712.05877.pdf. [Accessed: 16- Mar- 2019]

[9]"Open Images Dataset V4". [Online]. Available: https://storage.googleapis.com/openimages/web/index.html. [Accessed: 16- Mar- 2019]

[10]L. Crockett, R. Elliott, M. Enderwitz and R. Stewart, *The Zynq book*, 1st ed. Glasgow: Strathclyde Academic Media, 2014.

[11]"PYNQ", *Media.readthedocs.org*, 2019. [Online]. Available: https://media.readthedocs.org/pdf/pynq/latest/pynq.pdf. [Accessed: 16- Mar- 2019].

[12]H. Rhody, "Edge Detection", *Cis.rit.edu*, 2019. [Online]. Available: https://www.cis.rit.edu/people/faculty/rhody/EdgeDetection.htm. [Accessed: 16- Mar- 2019].

[13]A. Kouris, C. Bouganis, EIE 1st YEAR PROJECT REVISION and Q&A Session [Online]. Available: https://bb.imperial.ac.uk/bbcswebdav/pid-1525517-dt-content-rid-5017947_1/courses/DSS-EE1_IPRJ-18_19/tutorial-2019%281%29.pdf [Accessed: 16- Mar- 2019].

[14]"9 road signs for you life". [Online]. Available: https://huffpost.com. [Accessed: 16- Mar- 2019]

[15]"Calorie Mama Food AI". [Online]. Available: https://www.caloriemama.com. [Accessed: 16- Mar- 2019]

Jongejan, J. and Johnson, I. (2019). googlecreativelab/quickdraw-dataset. [online] GitHub. Available at: https://github.com/googlecreativelab/quickdraw-dataset [Accessed 17 May 2019].