Weakly Supervised Object Localization on FPGA

Vinam Arora
Georgia Institute of Technology
Atlanta, U.S.A
vinam@gatech.edu

Meghana Mallikarjuna
Georgia Institute of Technology
Atlanta, U.S.A
mmallikarjuna6@gatech.edu

Sreemanth Prathipati
Georgia Institute of Technology
Atlanta, U.S.A
sreemanth@gatech.edu

***Abstract*—*Weakly Supervised Object Localization* (WSOL) involves detecting the location of an object present within an image, only with a small amount of labeled data. Grad-CAM is a localization algorithm that can generate a bounding box on objects in images using only a network that was trained for image classification. In this project, we will implement GradCAM for WSOL on FPGAs and compare the performance with a PyTorch-based implementation.**

***Index Terms*—field programmable gate arrays, object detection, neural networks, WSOL**

I. Introduction

Weakly supervised object localization is a computer vision technique that involves training an algorithm to identify the presence and location of an object within an image, using only a small amount of labeled data. FPGAs (Field-Programmable Gate Array) can be used to accelerate the processing of these algorithms, leading to faster and more efficient object detection. In this project, we will implement GradCAM, which uses gradients flowing into the final convolutional layer to produce a coarse heatmap of important regions in the image for assigning it to the particular class. We will compare the performance of our FPGA implementation against a PyTorch-based GPU implementation.

Fig. 1: Example of the object localization task

II. Problem Description

The problem of object localization involves detecting and locating the bounding box of an object within an image. Traditional approaches to this problem involve using large datasets of labeled images to train deep learning models in a supervised fashion, which can then accurately detect and locate objects. However, obtaining these labeled datasets can be time-consuming and expensive.

Weakly Supervised Object Localization (WSOL) offers a solution to this problem by using a smaller amount of labeled data, often in the form of image-level labels, combined with additional, unlabeled data to train models that can still achieve accurate object detection and localization.

Using FPGAs to accelerate these algorithms can improve their speed and efficiency especially compared to the state-of-the-art implementations that run on GPUs, making them more practical for real-time applications.

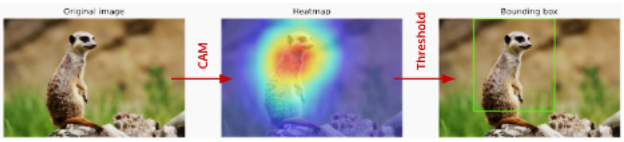

Fig. 2: Standard CAM-based WSOL pipeline

III. Related Work

Our work mainly draws on recent advancements in Weakly Supervised Object Localization on CAM-based algorithms. [3] gives a comprehensive survey of classic and deep learning-based approaches, publicly available datasets, and standard evaluation metrics used for weakly supervised object localization and detection, along with a discussion of key challenges, advantages, relationships, applications, and potential future directions for this field. We are interested in WSOL in the context of CNNs, which reduces the need for human annotation and makes training more efficient than the traditional Object Localization methods that use fully supervised learning approaches. Previous works have implemented WSOL using many CAM-based algorithms which are all on GPU/CPU. Our work aims at obtaining at-par results with an implementation on an FPGA, aiming at a better performance as a whole.

Most relevant to our work is the paper Grad-CAM: Visual Explanations from Deep Networks via Gradient-based Localization [2]. A drawback of CAM is that it requires feature maps to directly precede softmax layers, so it is only applicable to a particular kind of CNN architectures performing global average pooling over convolutional maps immediately prior to prediction. Such methods may achieve inferior accuracies compared to the generalized networks on some tasks. Grad-CAM on the other hand provides a way to combine the feature maps with the gradient signal that does not need any modification in the network architecture.

Other improvements include the LayerCAM which extracts hierarchical class activation maps for localization [1]. The class activation maps generated by LayerCAM tend to cover more object regions than the CAM-based methods which capture information from just the final convolutional layer. [1] gives a comparison of localization accuracy obtained from different CAM-based methods, suggesting LayerCAM has a higher accuracy.

We aim at implementing WSOL algorithms on FPGAs which support customization and are highly power-efficient devices. Our goal is to implement a comparable model and give a comparison between the accuracy obtained with the software developed on a GPU with respect to an onboard implementation.

IV. Your Proposed Approach

We want to implement a WSOL bounding-box pipeline on the Pynq Z2 FPGA board. We will initially aim at using a ResNet-18 trained on ImageNet1K dataset as our base model to perform CAM with. We use this dataset/model combination since it is easy to find a pre-trained model for this dataset. The model complexity should be small enough to fit within the resources of the Pynq Z2 board.

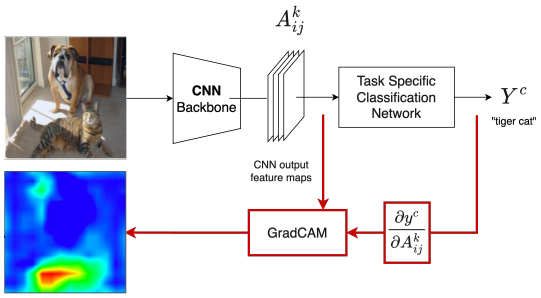

Fig. 3: Visual description of the Grad-CAM based WSOL pipeline

We will be starting with a GradCAM-based approach to achieve bounded boxes as described in [2]. Figure 3 and as well as the steps given below describe this algorithm:

1) Perform a forward pass on the model and make a classification. Y^c is the output logit corresponding to the chosen class. Store A_{ij}^k, which is the output of the CNN backbone. Here k is the output feature map index, and i, j are the spatial indices. Also store all the activation values in the classification network (following the CNN backbone) which are necessary to perform the backpropagation in Step 2.
2) Compute the derivative of output logit wrt. the output feature map:
$$\frac{\partial Y^c}{\partial A_{ij}^k}$$
3) Compute importance of each feature map α_k^c at the output of the CNN backbone:
$$\alpha_k^c = \sum_i \sum_j \frac{\partial Y^c}{\partial A_{ij}^k}$$
4) Generate a coarse heatmap:
$$L_{Grad-CAM}^c = ReLU\left(\sum_k \alpha_k^c A^k\right)$$
5) Scale coarse map to input resolution using bi-linear interpolation.
6) Threshold and bounding box generation algorithm (TBD). This will not be implemented on the FPGA.

Figure 4 contains the workflow of our proposed approach. First, we will set up a software model using PyTorch and run this on a GPU to get golden data we can use later to verify our FPGA implementation. We will also obtain performance metrics such as latency and bandwidth from these GPU runs for comparison later. Following this, we will work on the HLS implementation of the different logical components of our project - namely the forward pass, backpropagation, and GradCAM. Once we are done with this and have verified our implementations, we will implement the design on an FPGA and extract performance metrics from this implementation which will be compared against our GPU numbers.

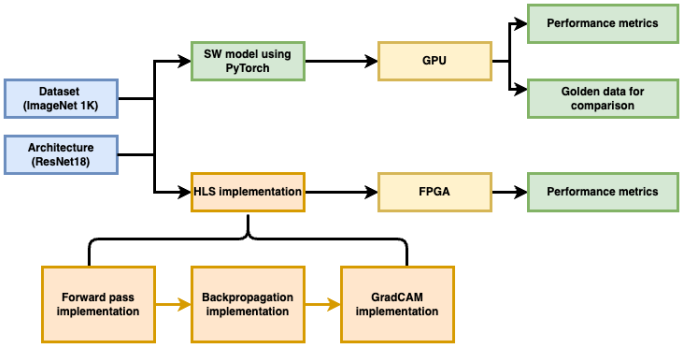

Fig. 4: Proposed workflow

V. Execution Plan

Detailed timeline:

03/06 - 03/13 :	Literature review
03/13 - 03/29 :	PyTorch model setup, quantization
03/29 - 04/20 :	CModel setup and HLS implementation
04/20 - 05/02 :	Further optimization and extensions
04/24 - 05/02 :	Report writing

Work Attribution: Below is the per-person breakdown of tasks planned.

- Vinam:
  - Literature review
  - PyTorch model setup
  - Forward pass HLS implementation
  - Integration, testbench development, synthesis
  - Further extensions
  - Collecting results and report writing
- Meghana:

- Literature review
- Model quantization and parameter extraction
- CAM HLS implementation
- Integration, testbench development, synthesis
- Further extensions
- Collecting results and report writing

- Sreemanth:
  - Literature review
  - PyTorch model setup
  - Backpropagation HLS implementation
  - Integration, testbench development, synthesis
  - Further extensions
  - Collecting results and report writing

VI. Mid-term Update

A. Resnet18 Architecture

Resnet18 includes five convolutional stages(see Figure 5 for more details) including convolutional layers, batch normalization layers, and fully connected layers.

Layer Name	Output Size	ResNet-18
conv1	$112 \times 112 \times 64$	7×7, 64, stride 2
conv2_x	$56 \times 56 \times 64$	3×3 max pool, stride 2
		$\begin{bmatrix} 3 \times 3, 64 \\ 3 \times 3, 64 \end{bmatrix} \times 2$
conv3_x	$28 \times 28 \times 128$	$\begin{bmatrix} 3 \times 3, 128 \\ 3 \times 3, 128 \end{bmatrix} \times 2$
conv4_x	$14 \times 14 \times 256$	$\begin{bmatrix} 3 \times 3, 256 \\ 3 \times 3, 256 \end{bmatrix} \times 2$
conv5_x	$7 \times 7 \times 512$	$\begin{bmatrix} 3 \times 3, 512 \\ 3 \times 3, 512 \end{bmatrix} \times 2$
average pool	$1 \times 1 \times 512$	7×7 average pool
fully connected	1000	512×1000 fully connections
softmax	1000	

Fig. 5: ResNet18 layers

B. Python model

We implemented a Python model of Grad CAM using PyTorch to get golden data to verify the correctness of our design and to later extract performance metrics while running on a GPU. We used pre-trained Resnet18 weights which were obtained by training on the ImageNet1K dataset. We registered a backward hook on the output of the last convolutional layer to store the gradients during backpropagation. We then used these activations to display obtain the gradCAM score as mentioned above and to display the corresponding heat map superimposed over the input image. Figure 6 shows a sample output of the python model.

We also used PyTorch to store activations after every ResNet18 layer, and use another Python script to compare the activations stored by both CModel and PyTorch. We also used another PyTorch script to dump per-layer biases and weights with batch normalization absorbed in these values.

Fig. 6: Sample heatmap image output from the Python model

C. C model

In C model implementation, we used the fused trained software model weights as the input. We used the following basic building blocks to build our resnet18 model.

1) Conv block: The Convolution block is responsible for filtering the input feature map and generating a summary of the detected features within it. This is achieved through a series of MAC operations, based on the input kernel size, stride, and padding used in the architecture. Additionally, a Rectified Linear Unit (ReLU) is applied to each computation within the block. We use the fused weights consisting of both convolution and batch-norm from the python model, due to which there is no implementation of batch-norm layer in our C model.

2) Maxpool: The maxpool block is used to compute the maximum value in each window and reduce the spatial dimension of the conv block. We use 2D tiling to reuse the resources and optimize the latency and LUT usage.

3) AveragePooling: While maxpool block computes the max value in each window, the average pool calculates the average of all the values for each patch in a feature map.

4) Linear-FC: We then flatten the feature map obtained from the sequential layers above to get a 1D array.

We get the classification score and compare with the one achieved from our software model.

This summarizes the forward pass. Grad-CAM is supposed to be implemented on top of this model.

D. CAM layer

Since a CAM implementation in ResNet18 architecture is equivalent to implementing GRAD-CAM [2], we introduce a CAM layer wherein we take the weights of the output and activations of the last convolution layer as inputs. The CAM layer output is a 7×7 matrix, which constitutes the heatmap.

E. Bilinear Interpolation

We implement the Bilinear interpolation block to resize the heatmap to the original input image dimensions. We then

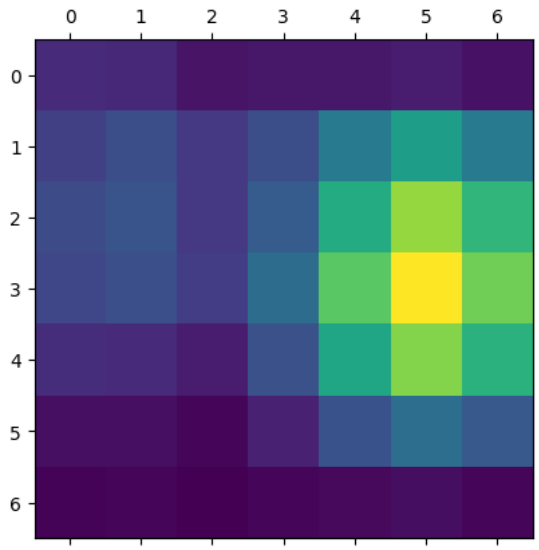

Fig. 7: 7×7 Heatmap

compare it with our python model output visually. Figure. 9 shows the superimposed heatmap for the test image.

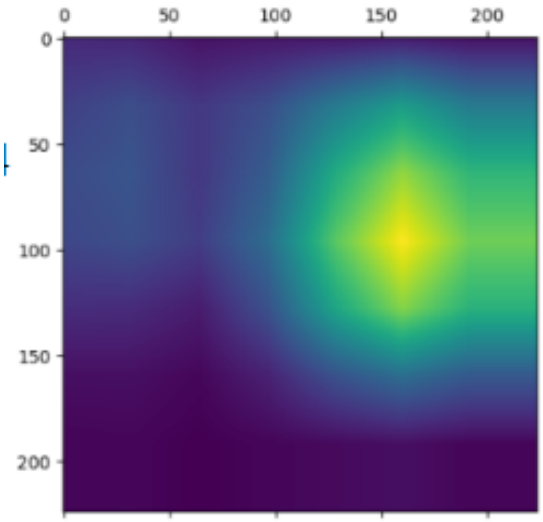

Fig. 8: Resized Heatmap

F. HLS Implementation

Different layers of the model are scheduled on the compute blocks sequentially. The output of the layer is written back to the output buffers and then reloaded into the input buffers while executing the next layer. Synthesizing the whole model gives us an initial estimate of metrics for starters. This will be further tuned to optimize the resource usability and latency.

VII. Mid-term Results

We achieved an MSE of 1.24×10^{-11} for the complete forward pass implementation with floating-point datatypes. The

Fig. 9: Superimposed Image

MSEs at individual layers are given in Table I. Table II shows the resource utilization metrics achieved from synthesizing the unoptimized forward propagation in Vitis HLS.

Layer	MSE
`conv1`	1.42×10^{-14}
`max pool`	1.81×10^{-14}
`conv2_x`	1.73×10^{-12}
`conv3_x`	7.49×10^{-13}
`conv4_x`	2.78×10^{-13}
`conv5_x`	1.23×10^{-11}
`average pool`	1.97×10^{-12}
`fully connected`	1.24×10^{-11}

TABLE I: MSE at layer outputs

Field	Value
Latency	72.34s
DSP	80 (36%)
FF	20908 (19%)
LUT	47451 (89%)
BRAM	23740 (8382%)

TABLE II: Resource Utilization

VIII. Final Hardware Architecture

Due to severe limitations of resources available on our FPGA board (PYNQ-Z2), a lot of thought went into refactoring the ResNet18 model for it.

A. Convolution Layers

One big challenge was handling convolution layers with various input shapes, strides, and the presence/absence of residual connections. These are listed in Table III, where 's' is the stride value for that layer, and 'r' indicates the presence of a residual connection at the output. (We wanted to fuse convolutions with residual connections where we could maximize hardware reuse)

Our architecture maps all convolution layers to 3 hardware kernels described in the following subsections:

1) `conv1`*:* A fixed input-size 7×7 convolution kernel designed much like Lab 2 of this course. The base kernel has a size of $8 \times 3 \times 7 \times 7$ and is repeatedly used $(64/8) \times (224/7) \times (224/7)$ times to create the output.

Using complete partitioning along the output buffer depth dimension, this layer introduces compute parallelism of 8. Vitis results show that this layer has a latency of $202ms$.

2) `conv_ds`*:* This kernel performs all the 1×1 downsampling operations. Since the input/output dimensions will be different for this layer, we use a 3D input tiling approach (as opposed to 2D input tiling in `conv1`). That is, the input is tiled across rows, columns, and depth.

The base kernel is $16 \times 16 \times 14 \times 14$ in size, and special optimizations were made since this layer has to perform 1×1 convolutions. This kernel has compute parallelism of 16.

3) `conv_3x3`*:* Similar to the downsampling kernel, this kernel also performs 3D input tiling. The base kernel is $16 \times 64 \times 3 \times 3$ in size and has a compute parallelism of 64. In addition, this kernel accommodates a programmable stride of 1 or 2, and fuses the optional addition of residual values.

Vitis results show that this layer takes $\sim 90ms$ for occurrences with a stride of 1, and $\sim 45ms$ for occurrences with a stride of 2.

Layer	Kernel Type	Input shape	Hardware mapping
conv1	7×7, s=1	$3 \times 224 \times 224$	conv1
conv2_00	3×3, s=1	$64 \times 56 \times 56$	conv_3x3
conv2_01	3×3, s=1, r	$64 \times 56 \times 56$	conv_3x3
conv2_10	3×3, s=1	$64 \times 56 \times 56$	conv_3x3
conv2_11	3×3, s=1, r	$64 \times 56 \times 56$	conv_3x3
conv3_ds	1×1, s=2	$64 \times 56 \times 56$	conv_ds
conv3_00	3×3, s=2	$64 \times 56 \times 56$	conv_3x3
conv3_01	3×3, s=1, r	$128 \times 28 \times 28$	conv_3x3
conv3_10	3×3, s=1	$128 \times 28 \times 28$	conv_3x3
conv3_11	3×3, s=1, r	$128 \times 28 \times 28$	conv_3x3
conv4_ds	1×1, s=2	$128 \times 28 \times 28$	conv_ds
conv4_00	3×3, s=2	$128 \times 28 \times 28$	conv_3x3
conv4_01	3×3, s=1, r	$256 \times 14 \times 14$	conv_3x3
conv4_10	3×3, s=1	$256 \times 14 \times 14$	conv_3x3
conv4_11	3×3, s=1, r	$256 \times 14 \times 14$	conv_3x3
conv5_ds	1×1, s=2	$256 \times 14 \times 14$	conv_ds
conv5_00	3×3, s=2	$256 \times 14 \times 14$	conv_3x3
conv5_01	3×3, s=1, r	$512 \times 7 \times 7$	conv_3x3
conv5_10	3×3, s=1	$512 \times 7 \times 7$	conv_3x3
conv5_11	3×3, s=1, r	$512 \times 7 \times 7$	conv_3x3

TABLE III: Different convolution layers of ResNet18

B. Maxpool

Our maxpool hardware kernel operates on $64 \times 14 \times 14$ tiles and performs 3×3 maxpooling with a stride of 2. No parallelism was introduced since this layer has much lower latency as compared to convolution. Vitis results show that this layer has a latency of $22ms$.

C. Average Pool

Our average pool kernel performs 1D tiling of the input along the depth dimension. A tile size of $16 \times 7 \times 7$ is chosen and no parallelism is introduced. Vitis results show a latency of $1.13ms$ for this layer.

D. Fully connected

This layer performs tiling on the output depth dimension. A tile size of 16×512 was chosen. Vitis results show a latency of $10.1ms$ for this layer.

E. CAM

The CAM block operates on the activations from the last convolution layer and the weights of fully connected layer. There was no tiling done as the block already had a much lower latency which was not comparable to others. Vitis latency for this block was $2.3ms$.

IX. Vitis Results

The results of Vitis HLS synthesis are shown in Table IV. Vitis was not able to provide a reasonable estimate of latency because of all the dynamic for-loops in our implementation. The reported latency was $100s$, which did not make sense since the layers take about 100ms individually. So, we calculated it using a spreadsheet (Fig. 10). This is the latency number provided in the table.

	OUTD	OUTW	OUTH	IND	TILE K	TILE L	TILE R	TILE C	TOTAL TILES	Time (ms)
L1.0 C1	64	56	56	64	4	1	8	8	256	89.6
L1.0 C2	64	56	56	64	4	1	8	8	256	89.6
L1.1 C1	64	56	56	64	4	1	8	8	256	89.6
L1.1 C1	64	56	56	64	4	1	8	8	256	89.6
L2.0 C1	128	28	28	64	8	1	4	4	128	44.8
L2.0 C2	128	28	28	128	8	2	4	4	256	89.6
L2.1 C1	128	28	28	128	8	2	4	4	256	89.6
L2.1 C2	128	28	28	128	8	2	4	4	256	89.6
L3.0 C1	256	14	14	128	16	2	2	2	128	44.8
L3.0 C2	256	14	14	256	16	4	2	2	256	89.6
L3.1 C1	256	14	14	256	16	4	2	2	256	89.6
L3.1 C2	256	14	14	256	16	4	2	2	256	89.6
L4.0 C1	512	7	7	256	32	4	1	1	128	44.8
L4.0 C2	512	7	7	512	32	8	1	1	256	89.6
L4.1 C1	512	7	7	512	32	8	1	1	256	89.6
L4.1 C2	512	7	7	512	32	8	1	1	256	89.6
									CONV1	201
									MAXPOOL	22
								TOTAL	3712	1522.2 ms

Fig. 10: Our calculation of latency

Field	Value
Latency	1.52s
DSP	57 (25%)
FF	35821 (33%)
LUT	52509 (98%)
BRAM	120 (42%)

TABLE IV: Vitis HLS Synthesis results

X. On Board results

Upon running the synthesized design on-board, we observed a latency of $5.28s$ as shown in Fig. 11. The synthesized design is shown in Fig. 12

```
start_time = time.time()

dut.register_map.CTRL.AP_START = 1
dut.register_map.CTRL[4] = 1
while not dut.register_map.CTRL.AP_DONE:
    end_time = time.time()
    duration = end_time - start_time
    if (duration < 10):
        pass
    else:
        break

end_time = time.time()
duration = end_time - start_time
print(f'Kernel completed in {duration * 1000:.2f}ms')
```

```
Kernel completed in 5288.92ms
```

Fig. 11: On-board latency measurement

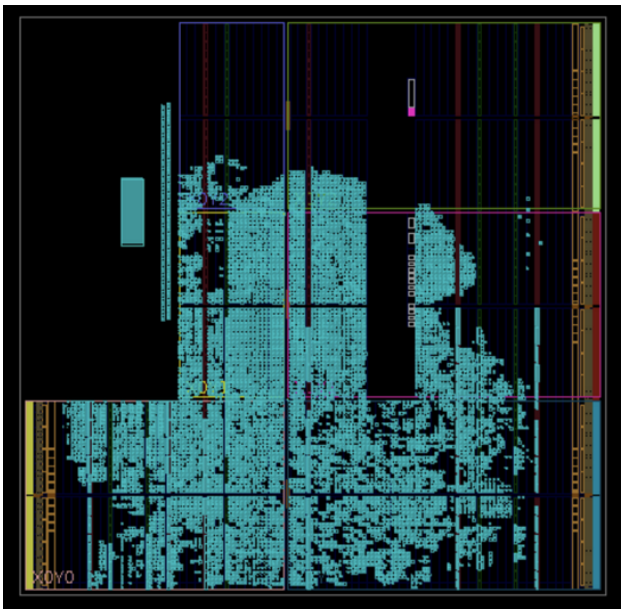

Fig. 12: Synthesized design

XI. Comparison with baseline

We used our laptop CPUs to get baseline numbers, as shown in Table V. Since our FPGA is only operating at 100MHz, a fair comparison would look at the right-most column of this table.

As we can see, our latency numbers show that our design performs worse than modern CPUs by a factor to $3\times$ or $9\times$ for M1 and M1 Pro respectively. This can largely be attributed to the lack of layer-wise pipelining in our design, and has been addressed in Sec XII.

Hardware	Operating freq	Latency	Latency @ 100MHz
Apple M1 CPU	3GHz	70ms/image	2.1s/image
Apple M1 Pro CPU	3GHz	20ms/image	0.6s/image

TABLE V: Baseline latencies

XII. Experiments and Future Work

It is possible to pipeline the ResNet18 implementation across layers. This would not bring the latency down, but increase the throughput from 5s/image to an estimated 200ms/image. This estimate comes from the fact that the `conv1` layer is the slowest one, and thus, would be the limiting factor. Pipelining would also require remapping the DRAM addresses of activation layers so that we can perform IO in a ping-pong buffer fashion.

Due to the constraints of time, we were not able to implement the logic that computes bounding boxes from the saliency map output. Implementing that would be part of the future work, and would complete the WSOL pipeline.

We were not able to implement image resizing on FPGA. This is the hardware that resizes the 7×7 heatmap to the original image size (224×224) using bilinear interpolation. Since there is a high degree of parallelism in this operation, it is well suited to be performed on an FPGA.

Even though Vitis reported a LUT usage of 98%, synthesis results told otherwise. Hence, our FPGA does have resources left that we can use to optimize our design further. So, future work involves performing more design space exploration to reach better latencies.

The PYNQ-Z2 FPGA development board might not be the most suited for such a large and complex hardware project. So, moving to another FPGA board (like the ZCU board) will be a better way forward. Early tests with this idea showed that increasing parallelism by $4\times$ only leads to a latency decrement of $2\times$. Thus, we would be limited by load/store latencies at that point and would need to consider double buffering inside the convolution kernels.

XIII. Implementation

Our implementation is available for anyone to freely use at https://github.com/vinamarora8/gradcam-fpga

References

[1] P.-T. Jiang, C.-B. Zhang, Q. Hou, M.-M. Cheng, and Y. Wei, "Layercam: Exploring hierarchical class activation maps for localization," *IEEE Transactions on Image Processing*, vol. 30, pp. 5875–5888, 2021.

[2] R. R. Selvaraju, A. Das, R. Vedantam, M. Cogswell, D. Parikh, and D. Batra, "Grad-cam: Visual explanations from deep networks via gradient-based localization," *arXiv:1610.02391*, 2016.

[3] D. Zhang, J. Han, G. Cheng, and M.-H. Yang, "Weakly supervised object localization and detection: A survey," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 44, no. 9, pp. 5866–5885, 2022.