Acceleration of YOLOv4-tiny Object Detection Algorithm using HLS

Anusha Sumbetla
Electrical and Computer Engineering
Georgia Institute of Technology
asumbetla3@gatech.edu

Sibi Renganatth Sudhakar
Electrical and Computer Engineering
Georgia Institute of Technology
ssudhakar32@gatech.edu

Viswanath Kondepudi
Electrical and Computer Engineering
Georgia Institute of Technology
vkondepudi3@gatech.edu

***Abstract*—This is a progress report for our FPGA project titled Acceleration of YOLOv4-tiny Object Detection Algorithm using HLS. In this project, we want to tackle the issue of speeding up the state-of-the-art object detection algorithm YOLOv4 using the Pynq FPGA which we have in our labs. We will be using methods like pipelining the loops, partitioning the arrays, streaming and buffering to achieve speedup in the backbone structure of the YOLOv4-tiny algorithm.**

***Index Terms*—Deep Learning, Convolution Neural Networks, Accelerator, FPGA, YOLOv4-tiny, Vitis HLS**

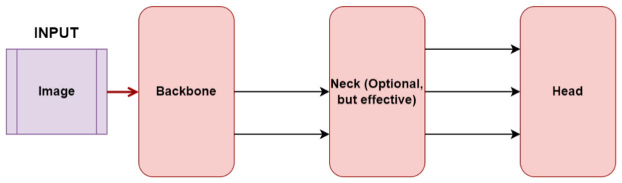


Fig. 1. Structure of YOLOv4-tiny.

I. Introduction

Object Detection has become the most widely used application in recent times. From self-driving cars to biological discoveries, the need for good and fast object detection has become inevitable. Object detection algorithms find great use in Advanced Driver Assistance Systems(ADAS), medical imaging and crowd surveillance. Earlier algorithms based on R-CNN [1] involve two stages - the first stage being the region selection which finds the possible objects/candidates that contain an object and a deep neural network performing classification of the selected regions. These algorithms are usually computation-heavy and have high detection latency. This makes them unusable or unsuitable for low-power embedded devices.

Hence, this brought forward the YOLO [2] algorithm, a one-step object detection algorithm. The algorithm works by using DNN algorithms to predict the bounding boxes around the objects and classify the objects. There have been multiple iterations of the YOLO algorithm, with each version an improvement of its predecessor. The YOLO algorithms have demonstrated both lower latencies and better classification precision over similar algorithms. There are also scaled versions of YOLO algorithms that can be run on mobile devices and embedded systems. The YOLOv4-tiny [3] is a faster version of YOLOv4 which uses multiple convolution layers to detect multiple objects in a single frame. The YOLOv4-tiny is also power efficient due to its smaller size and can be used in real-time applications and low-power devices.

II. Problem Description

A. Description of the YOLOv4-tiny

The YOLOv4-tiny is a convolution-based object detector. It is composed of three main parts (Fig.1) - the Backbone, the Neck and the Head. The input image is fed into the Backbone, also called a feature extractor, is a convolution neural network that pools image pixels to form features at different granularities. The YOLO neck (feature aggregator) takes in feature maps from different stages of the backbone and combines and mixes them before sending them out to the head. The Head which is the object predictor is the part that draws the bounding boxes and makes the class prediction.

Implementing the YOLOv4-tiny algorithm on embedded systems will lead to various improvements/applications for mobile phones. Hence we look to implement the YOLOv4-tiny algorithm on the PYNQ FPGA board and work on speeding up the algorithm. This is really important as YOLOv4-tiny algorithm is compute-heavy with multiple convolution structures.

The backbone structure of the YOLOv4-tiny is based on the Tiny CSPdarknet53 architecture and has multiple convolution neural networks. The large number of layers and parameters in this part makes it a complex and computation-heavy part of the YOLOv4-tiny.

Hence, deploying YOLOv4-tiny algorithm in a low-power FPGA device would be challenging due to resource limitations and off-chip memory utilization. We aim to analyze and reach a solution that would achieve lower latencies and meet the resource constraints.

III. Related Work

There have been several works deploying different versions of YOLO algorithms on FPGAs. The authors in [4] implement a fixed-point quantized FPGA architecture for YOLOv4-tiny algorithm to reduce the bandwidth requirement and improve the speed of the inference model. Batch normalization and convolution models were combined together and quantization is performed to get the best out of the FPGA.

Similar work on YOLOv4-tiny architecture [5] involves a high resource intensive FPGA to detect coal gangue from a given set of pictures. This also involves fixed point quantization of the weights and normalization of batch normalizing factors with the convolution outputs to accelerate the model.

Another work [6] involved a dynamic 16-bit fixed point number quantization and achieved a 4 time speedup for the same YOLOv4-tiny network.

We also observed that the works on earlier YOLO tiny algorithms like [7], [8] and [9] on YOLOv3-tiny used similar methods of fusing the convolution and batch normalization processes and quantization of weight outputs. It is also noted that specific multiplication methods like General Matrix Matrix Multiplication are adopted to speed up the process.

In general, we observe that the weights can be stored on-chip or off-chip depending on the number of convolutions involved. In the case of YOLOv4-tiny, there are both versions of on-chip layer parameters that require high-performance FPGAs

IV. Detailed description of Layers of Backbone Structure in YOLO-v4 Tiny

Through compressing the network structure and simplifying feature extraction, YOLOv4-tiny makes the network lightweight. The process may reduce algorithm detection accuracy, but it greatly increases detection speed and still provides high accuracy in object detection.

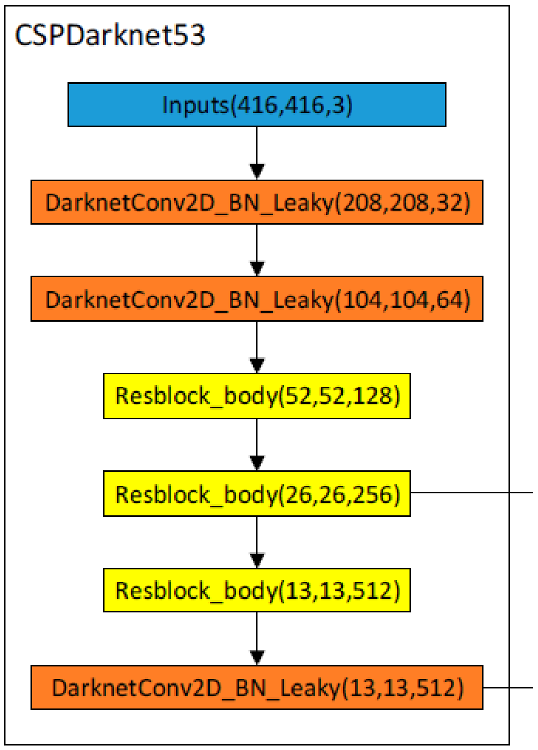

CSPDarknet53
Inputs(416,416,3)
DarknetConv2D_BN_Leaky(208,208,32)
DarknetConv2D_BN_Leaky(104,104,64)
Resblock_body(52,52,128)
Resblock_body(26,26,256)
Resblock_body(13,13,512)
DarknetConv2D_BN_Leaky(13,13,512)

Fig. 2. Detailed Backbone structure of YOLOv4-tiny.

YOLOv4 Tiny backbone network consists of several convolutional layers that extract features from input images. Backbone networks detect low-level features, such as edges and corners, which are then used to detect higher-level features, such as shapes and textures.

YOLOv4 tiny backbone has total of 15 convolutional layers, 3 maxpool layers along with activation function. With a dimension of (3,416,416), it takes an input image and converts it into a dimension of (512,13,13).

The backbone architecture of YOLOv4-Tiny has multiple blocks that are repeated throughout the network to extract features from an input image at various scales and resolutions. These blocks include DarknetConv2D blocks and Resblock body.

A. *DarknetConv2D Block*

This block consists of convolutional layer followed by batch normalization and leaky ReLU activation. We have implemented first two DarknetConv2D blocks as part of the project which takes (3,416,416) size image as input.

The first block consists of a convolutional layer which applies 32 filters that are of size 3x3. These are applied with a stride of 2 in both horizontal and vertical directions. Also, a padding 1x1 is applied to maintain the spatial dimension of the input. The resulting feature map output from the convolutional layer is of dimensions (32,208,208). Batch normalization followed by Leaky ReLU is then applied on Convolution network output to improve the stability and performance. The resulting output is then fed as an input to the next block which is also an Darknetconv2D block. The second block applies 64 filters that are of size 3x3 with stride 2 and padding 1 that results in the output of dimensions (64,104,104).

B. *Resblock Body*

ResBlock enables the network to capture features at different scales and improve the flow of gradients. It consists of 4 convolutional layers along with batch normalization, leaky ReLU and 2 concatenation layers, a cross stage partial connection layer and a Max pool function.Maxpool is used to reduce the computational complexity.

The input from previous Conv block which is of dimension (64,104,104) is passed in to the first layer of Resblock which applies 64 filters that are of size 3x3 with stride 1 and padding 1. The resulting output feature map is also of dimensions (64,104,104) because of stride 1. Each convolution layer also applies batch normalization followed by Leaky ReLU on the Convolution network output to improve the stability. The generated output is then fed to CSP layer which splits the input to two groups and sends only one group to the next convolution layer. The resulting dimensions of the output after the split would be (32,104,104) which is fed as an input to the second convolutional layer.

The second convolutional layer applies 32 filters that are of size 3x3 with stride 1 and padding 1. This results in the output of dimension (32,104,104) which is then fed to the third convolutional layer. This layer also applies 32 filters that are of size 3x3 with stride 1 and padding 1 that results in the output with dimension (32,104,104). This output is concatenated with the output of the second convolutional layer and a output of dimensions (64,104,104) is obtained.

The concatenated output is then sent to the fourth and final convolutional layer which applies 64 filters that are of size 1x1 with stride 1 that results in the output with dimension (64x104x104). This is then concatenated with the first convolutional layer and a ouptut of (128,104,104) is generated. Finally, Maxpool is applied and the dimensions of feature map are further reduced to (128,52,52).

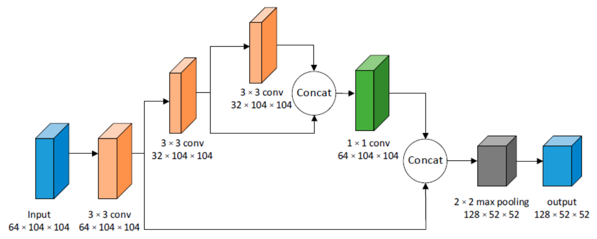


Fig. 3. Resblock structure of YOLOv4-tiny.

V. Proposed Approach

Our plan is to combine the advantages of fixed-point quantization and batch normalization fusion into the existing YOLOv4-tiny algorithms and evaluate the speedup achieved. Previous approaches have only used either of the above optimizations. Our approach will try to combine the best of both worlds and achieve a higher speedup.

The backbone of YOLOv4-tiny comprises 15 convolution networks, with each network having its own kernel and stride lengths. DarknetConv2D layer involves 2D convolution followed by a layer of batch normalization and leaky ReLU functions. So, our approach will focus on fusing the batch normalization and leaky ReLU layer outputs with the convolution layer weights.

Following this, our Resblock layer involves four layers of convolution, with each layer input being either the concatenation of the original input and the previous convolution layer output or the previous layer input. This would require pipeline and various structural optimizations in HLS.

So, our plan of action involves implementing the given YOLOv4-tiny architecture as a PyTorch model and achieve golden outputs using the MS-Coco dataset. Following this, we aim to achieve a golden C++ model mirroring the golden model behavior and storing the layer weights and outputs for use in the FPGA systems.

We will next focus on the quantization of the layer weights and outputs for use in the PYNQ FPGA board. Using the golden C++ model, we build upon the HLS model and run synthesis to view the unoptimized code performance. Following this, we optimize the HLS code using pipelining, loop reordering, loop tiling and ping pong buffers present in the PYNQ board to achieve speedup.

VI. Execution Plan

Phase 1: To conduct a thorough literature review of the topic and understanding the multiple blocks to be implemented in the algorithm. Analyzing the key concepts and ideas presented in the literature to gain a comprehensive understanding of the research problem. Identifying the main challenges and limitations associated with the algorithm, as highlighted in the literature.

Phase2: Using PyTorch to implement the algorithm. Developing the code for each block of the algorithm based on the insights gained from the literature review. Testing the implementation to ensure that it is functioning as expected.

Phase 3: Translating the Python implementation to C++. Ensuring that the C++ implementation is accurate and free from errors. Checking for synthesizability to ensure that the implementation can be synthesized for a hardware implementation.

Phase 4: Optimizing and improving the latency of the algorithm by using hardware acceleration techniques. Testing the optimized implementation to ensure that it is faster and more efficient than the previous implementations. Documenting the results of the implementation and optimization processes for future reference.

Phase 1 will be handled by all the members together. This phase will help us understand what aspects of the architecture are worth spending efforts to speed up.

Phase 2 PyTorch implementation will be handled by Viswanath. The other members will assist him with the necessary inputs for the implementation.

Phase 3 will be handled concurrently with Phase 2. Sibi and Anusha will split the model. Sibi will work on the Resblock structure implementation. Anusha will work on the Darknetconvolution implementation. Viswanath will take care of the integration of the different modules.

Phase 4 will also see a split up of work. Anusha and Viswanath will work on the Resblock HLS implementation and optimization. Sibi will work on the DarknetConv HLS implementation. The team will entirely work on performing multiple runs of optimization and integrate the final HLS implementation that gives us appreciable speedup.

VII. Midterm Report

We have finished the implementation of Phase1 and Phase2 of our proposed plan. After a thorough background review of the literature present on YOLOv4-tiny algorithm, we decided to implement a few layers of the backbone structure in the timeframe we have for the project. We will be implementing the Darknet convolution + Batch Normalization and Leaky RelU layers followed by the Resblock body in both PyTorch and then in C++ so as to be able to synthesize it and optimize the latency.

A. *PyTorch Implementation*

We obtained the trained model weights from the Darknet database and extracted the weights corresponding to the input, kernel and output. Each layer output was separately extracted from PyTorch and saved. Batch normalization and Leaky RelU layer outputs were also extracted from this PyTorch mode. Since the first two layers are convolution layers, the same was repeated for the second layer as well.

B. *C++ Implementation*

In C++, we implemented the convolution layer and matched it with the output obtained from PyTorch. Next, we implemented the Batch Normalization in C++ and made sure this is matching with the PyTorch output as well. Finally, we proceeded to implement the Leaky Relu in C++ to check the final outputs of the first layer of the YOLOv4-tiny algorithm. We proceeded to repeat the same with the second layer parameters as well.

VIII. Final Work

A. *Pytorch Implementation*

We looked to implement the YOLOv4-Tiny backbone structure using PyTorch models and our initial part of the project involved finding existing implementations of such models. Then we decided to use a Darknet based PyTorch model which implemented the entire YOLOv4-Tiny architecture including the CSP blocks in the backbone structure.

We had to use the trained weights and ran an inference on the model with some test images. Once we had the model, the next step we approached was the extraction of individual convolution layers. Since there are 23 convolution layers involved in the backbone structure only, we decided to implement the first two layers of convolution followed by a CSP block.

The pattern followed in the YOLOv4-tiny convolution block is as follows: There is a convolution layer with a specific kernel size, padding and filter sizes. This is followed by a batch normalization layer that normalizes the convolution output. Following this is the LeakyReLU activation function that smoothens the negative values.

As mentioned earlier, we need to extract the inputs, outputs, weights and biases for each convolution layer. So, we make use of the Pytorch model to extract the individual weights and biases. We also extract the inputs and outputs of each convolution layer to verify against the HLS implementation of the individual layers.

1) Fusing of Batch Normalization and Convolution layers: Batch normalization involves the calculation of the mean and variance of the convolution output and normalizes the output by subtracting the mean and dividing the variance. However, it is impossible to calculate the running mean and variance of the convolution output from the FPGAs due to the unavailability of resources. So, we have to fuse the convolution and batch normalization layers into one single convolution layer comprising weights and bias.

Hence, we did a Python based code for fusing the layers together and generate the fused weights and bias. We also compared the performance of fusing by calculating the MSE between the original output and the output obtained using fused weights. The MSE's for the first 7 layers were around 10^{-12}.

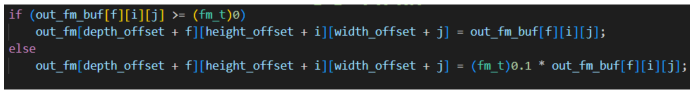
We implemented the fusing of the convolution and the batch normalization on the C++ as well. This is done to verify and ensure the sanity of the fused weights and biases.

B. *C++ Implementation*

For the C++ implementation of the YOLOv4-Tiny algorithm, we used the base code of our labs and modified it by adding additional functions wherever necessary. In YOLOv4-tiny backbone structure, we have multiple convolution layers. We also have concatenations, splits and maxpool layers which we have verified for functionality and optimized.

1) Batch-Normalization: Batch normalization is a technique commonly used in deep learning models, including YOLOv4 tiny, to improve the training and performance of neural networks. In YOLOv4 tiny, batch normalization is applied to the outputs of the convolutional layers. The batch normalization was implemented in C++ for verification purposes as we decided to go with fusing the weights with BN weights in the end. To implement this, the mean and variance of the convolution output was extracted. Using this, each individual element was normalized. To this the batch normalization weights and bias was added to obtain the final output after batch normalization. This helps to reduce the internal covariate shift, which is the change in the distribution of inputs to each layer that occurs during training. By reducing the internal covariate shift, batch normalization allows the network to learn more stable and effective representations of the input data. This output was then verified with that obtained from Pytorch after the batch normalization layer.

2) Leaky RelU: The Leaky ReLU activation function is a variant of the popular Rectified Linear Unit (ReLU) activation function that is commonly used in deep learning models. In YOLOv4 tiny, the Leaky ReLU activation function is used in the convolutional layers of the network.



```
if (out_fm_buf[f][i][j] >= (fm_t)0)
    out_fm[depth_offset + f][height_offset + i][width_offset + j] = out_fm_buf[f][i][j];
else
    out_fm[depth_offset + f][height_offset + i][width_offset + j] = (fm_t)0.1 * out_fm_buf[f][i][j];
```

Fig. 4. Leaky RelU used in YOLOv4-tiny.

In YOLOv4-tiny, the factor we have used is 0.1 , hence there is a small slope when the output is negative instead of a sharp drop to zero like in RelU function.

3) Mean Square Error Computation: The method used to verify our outputs obtained in C++ implementation was using the mean squared error. The error obtained by comparing the C++ implementation with the outputs obtained from Pytorch gave us the confidence that our implementation of C++ is functionally correct.

4) Splitting: In the YOLOv4-tiny, the CSP Layer comprises of a convolution layer where the output is split into two. One of these outputs goes as the input to the immediate layer and the other half goes and concatenates to the future convolution layers. This was implemented in C++ by using a for loop and assigning an output array with half the values of the input.

5) Concatenation: The C++ implementation of YOLOv4-tiny is simply done using a for loop and assigning the corresponding values from the two arrays that need to be concatenated to the output array. The concatenation is done along the depth, hence the output of concatenation has double the depth of its inputs.

6) Max-Pool Layer: The Max-Pool layer is a technique used to down-sample the feature-map produced by the previous layers. In our Layer 7, Max-Pooling is done by taking a 2X2 kernel and computing the maximum among those values in that square neighborhood. This operation has its own specific parameters to be considered, like stride, padding and kernel size. In the end, since the parameters in YOLOv4-tiny are 2X2 kernel size with a stride of 2, the output dimensions are reduced by half along the height and width.

7) Flow: To run the layers, we decided to create a Makefile which runs one layer at a time on the FPGA, optimized using Vitis HLS. The Layer1 is first run, the output of which is stored as a binary file, this is then fed into Layer2, again output from this goes into the CSP Layers and so on. We have verified using the C++ tool that all the codes run correctly one after the other even with inputs from the previous layers.

C. *Vitis HLS Optimization Techniques*

All the below optimization techniques are done so as to reduce the overall latency by parallelizing and pipe-lining the loops. By having parallel accesses, the Vitis HLS adds additional resources in such a way as to reduce the overall time taken for the functionality.

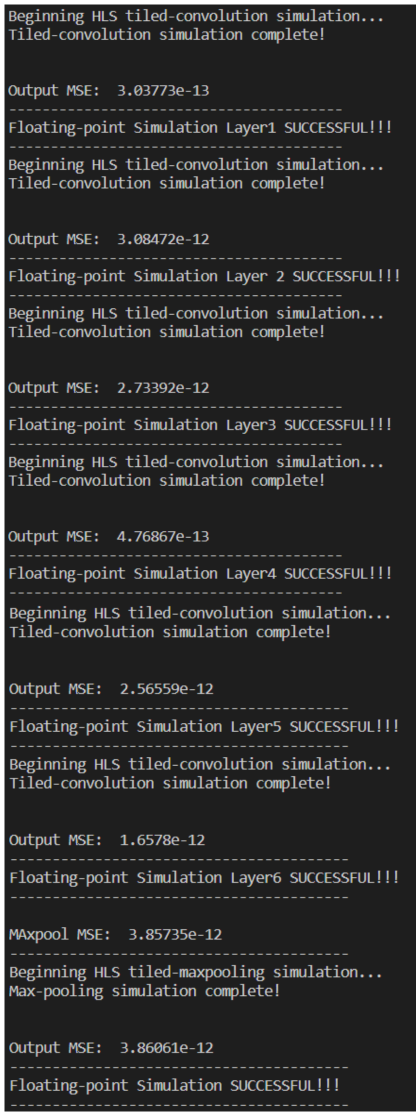
- Array partitioning: In this technique, large arrays are divided into smaller partitions to improve memory access time and overall performance. The dimension directive specifies which dimension the array should be partitioned. In our project, we have used both cyclic and complete array partitioning. In cyclic partitioning, the array is partitioned by a factor such that every element separated by that factor can be accessed simultaneously.
- Pipeline: Multiple loop iterations can be executed simultaneously with pipeline, increasing the performance of the hardware implementation. In the pipeline directive, the number of stages is specified in order to balance performance with resource utilization.
- Loop tiling: Large iteration space is divided into smaller portions to reduce resource utilization. This is achieved by reducing number of memory access and increasing parallelism.
- Loop reordering: The order of the nested loops is changed to improve performance and reduce resource utilization.
- Ping Pong Buffers: We used ping pong buffers to minimize the loop iterations and perform multiple convolution operations at the same time. This is done for the convolution layers having input depths greater than 16. Note, the buffers are extremely efficient to be used when there are multiple layers fused together. So, we have implemented ping-pong buffers to fuse convolution layers 1 and 2.

IX. Results and Observations

A. *MSE Computation*

Mean square is the parameter we have taken as a baseline to verify whether the output generated by our c++ code matches with the golden output. If the MSE for the generated output from each convolution layer is of the order 10^{-12}, we considered the floating point simulation as successful. We have successfully compiled and verified that all the outputs are matching with the golden outputs.

Below is the output of the console after running the make command.



```
Beginning HLS tiled-convolution simulation...
Tiled-convolution simulation complete!


Output MSE:  3.03773e-13
---------------------------------------
Floating-point Simulation Layer1 SUCCESSFUL!!!
---------------------------------------
Beginning HLS tiled-convolution simulation...
Tiled-convolution simulation complete!


Output MSE:  3.08472e-12
---------------------------------------
Floating-point Simulation Layer 2 SUCCESSFUL!!!
---------------------------------------
Beginning HLS tiled-convolution simulation...
Tiled-convolution simulation complete!


Output MSE:  2.73392e-12
---------------------------------------
Floating-point Simulation Layer3 SUCCESSFUL!!!
---------------------------------------
Beginning HLS tiled-convolution simulation...
Tiled-convolution simulation complete!


Output MSE:  4.76867e-13
---------------------------------------
Floating-point Simulation Layer4 SUCCESSFUL!!!
---------------------------------------
Beginning HLS tiled-convolution simulation...
Tiled-convolution simulation complete!


Output MSE:  2.56559e-12
---------------------------------------
Floating-point Simulation Layer5 SUCCESSFUL!!!
---------------------------------------
Beginning HLS tiled-convolution simulation...
Tiled-convolution simulation complete!


Output MSE:  1.6578e-12
---------------------------------------
Floating-point Simulation Layer6 SUCCESSFUL!!!
---------------------------------------

MAxpool MSE:  3.85735e-12
---------------------------------------
Beginning HLS tiled-maxpooling simulation...
Max-pooling simulation complete!


Output MSE:  3.86061e-12
---------------------------------------
Floating-point Simulation SUCCESSFUL!!!
---------------------------------------
```

Fig. 5. Console output

The MSE for Fixed Point simulations is relatively higher than the lab simulation. This is because the weights extracted from the PyTorch model require higher precision. So, the fixed point of $< 16, 3 >$ gives the MSE in the range of 1-2. We focused on varying the decimal bit representation for each individual layer and found that using the fixed point representation of $< 16, 6 >$ gives an optimal performance over the other representations.

Block	Layer	Fixed Point MSE
Conv2D	Layer 1	0.0629
Conv2D	Layer 2	0.105
Resblock	Layer 3	0.1508
	Layer 4	0.0338512
	Layer 5	0.06022
	Layer 6	0.0168029

Fig. 6. Fixed Point MSE for the 6 convolution layers

B. Latency

We have optimized using HLS optimization techniques on the convolution layers and reduced the combined latency of 3 blocks which includes 6 convolutional layers from 9.067 seconds to 0.505 seconds with a speed up of **18**.

Block	Layer	Latency in Seconds		
		Unoptimized	Optimized	Speed up
Conv2d	Layer1	0.409	0.029	14.1034
Conv2d	Layer2	2.056	0.12	17.1333
Resblock	Layer3	4.076	0.206	19.7864
	Layer4	1.026	0.056	18.3214
	Layer5	1.028	0.058	17.7241
	Layer6	0.472	0.036	13.1111
Total Latency		9.067	0.505	17.9545

Fig. 7. Latency for unoptimized and optimized convolutional layers

C. Resource Utilization

Each and every convolution layer is implemented separately and we verified that the resource utilization is below 100 percentage.

The HLS Optimization pragmas added resulted in increased resource utilization as the loops are now unrolled and executed in parallel. This helps in reducing the latency but increases the overall utilization.

D. HLS optimization of combining layers - Conv2D and Resblock

Once we had the relative performance for each individual layer, we decided to combine the first two layers of YOLOv4-Tiny backbone, that is, Conv2D layers. We then decided to combine the first two layers of Resblock architecture, which involves splitting of the input image into two sections and applying convolution. Currently, the first two layers of Conv2D are implemented. There is considerable work done in fusing

Optimized Resource Utilization					
Block	Layer	Utilisation %			
		BRAM	DSP	FF	LUT
Conv2d	Layer1	11	33	9	34
Conv2d	Layer2	7	18	7	24
Resblock	Layer3	5	17	5	23
	Layer4	5	17	5	23
	Layer5	5	17	5	23
	Layer6	4	17	4	20
	Layer7	9	2	2	21

Fig. 8. Optimized layer resource utilization

Unoptimized Resource Utilization					
Block	Layer	Utilization %			
		BRAM	DSP	FF	LUT
Conv2d	Layer1	2	9	5	21
Conv2d	Layer2	6	5	4	18
Resblock	Layer3	4	4	4	17
	Layer4	2	4	4	17
	Layer5	3	4	4	17
	Layer6	2	5	3	13
	Layer7	1	3	1	9

Fig. 9. Unoptimized layer resource utilization

the first two layers of the Resblock structure. Since there is time limitation, we have fused these two layers. On performing C++ simulations and HLS optimization, the results are as follows:

The MSE for fixed point and floating point optimizations have been similar to the individual layer MSEs.

Block	Floating Point MSE	Fixed Point MSE
Conv2D_Layer12	1.8206x10^-12	0.2464
CSP_Layer12	2.0319x10^-12	0.161401

Fig. 10. MSEs of Fused layers

Resource Utilization for the combined layers: The resource utilization achieved in fusing the layers are also considerably similar to the aggregate of the individual layers. In addition to this, the latency of the combined layers is comparable with the C++ implementation as well as the aggregate of individual layers.

Resource Utilization due to combining layers %					
Block	BRAM	DSP	FF	LUT	Latency
Conv2D_Layer12	15	16	10	32	2.347s
Resblock_Layer12	46	18	11	27	5.195s

Fig. 11. Resource Utilization and Latency from Layer Fusing

X. Future Work

There is a broader scope in the project where we can implement the entire CSP structure in the FPGA and use multiple iterations as in the backbone structure. Since the weights are taken from high precision, there is potential work needed in optimizing and quantizing the weights data to improve the performance in FPGAs.

The backbone structure of YOLOv4-Tiny can be further optimized using data streaming and aggressive loop optimizations. This can help us achieve the maximum speedup. The current combination and fusing of two layers each in Conv2D and Resnet blocks show us the promise of adding two to three layers on top of the existing implementations.

Also, the introduction of YOLOv5, YOLOv6 and YOLOv7 algorithms show a vast area of FPGA based acceleration for these object detection algorithms. Since each YOLO algorithm is an extension and architecture-based optimizations of the previous layers, it could mean that there is a wider scope of accelerating Machine Learning algorithms.

XI. Conclusion

Our project aimed at implementing a part of the YOLOv4-Tiny backbone structure and accelerate using FPGA. The current results and observations indicate that there is a wider scope and reach for FPGA to accelerate YOLO algorithms with baseline optimizations such as loop reordering, unrolling and array partitioning. We also observed that the fusing and combining multiple layers and accelerating through FPGA has been successful.

The baseline implementation of the inital 6 layers of the YOLOv4-Tiny algorithm in FPGA indicates that the latency achieved on the Vitis HLS simulations and the latency on board, considering pessimism, is lesser than running the same algorithm on a CPU/GPU.

References

[1] Ren, S., He, K., Girshick, R. and Sun, J., 2015. Faster r-cnn: Towards real-time object detection with region proposal networks. Advances in neural information processing systems, 28..

[2] Bochkovskiy, Alexey, Chien-Yao Wang, and Hong-Yuan Mark Liao. ”Yolov4: Optimal speed and accuracy of object detection.” arXiv preprint arXiv:2004.10934 (2020).

[3] Jiang, Zicong, et al. ”Real-time object detection method based on improved YOLOv4-tiny.” arXiv preprint arXiv:2011.04244 (2020).

[4] Fengxi Zhang et al 2022 J. Phys.: Conf. Ser. 2303 012032.

[5] Xu, Shanyong, et al. ”YOLOv4-Tiny-Based Coal Gangue Image Recognition and FPGA Implementation.” Micromachines 13.11 (2022): 1983.

[6] P. Li and C. Che, ”Mapping YOLOv4-Tiny on FPGA-Based DNN Accelerator by Using Dynamic Fixed-Point Method,” 2021 12th International Symposium on Parallel Architectures, Algorithms and Programming (PAAP), Xi’an, China, 2021, pp. 125-129, doi: 10.1109/PAAP54281.2021.9720468.

[7] T. Adiono, A. Putra, N. Sutisna, I. Syafalni and R. Mulyawan, ”Low Latency YOLOv3-Tiny Accelerator for Low-Cost FPGA Using General Matrix Multiplication Principle,” in IEEE Access, vol. 9, pp. 141890-141913, 2021, doi: 10.1109/ACCESS.2021.3120629

[8] Yu, Z., Bouganis, CS. (2020). A Parameterisable FPGA-Tailored Architecture for YOLOv3-Tiny. In: Rincon, F., Barba, J., So, H., Diniz, P., ´ Caba, J. (eds) Applied Reconfigurable Computing. Architectures, Tools, and Applications. ARC 2020. Lecture Notes in Computer Science(), vol 12083.

[9] A. Ahmad, M. A. Pasha and G. J. Raza, ”Accelerating Tiny YOLOv3 using FPGA-Based Hardware/Software Co-Design,” 2020 IEEE International Symposium on Circuits and Systems (ISCAS), 2020, pp. 1-5, doi: 10.1109/ISCAS45731.2020.9180843.