Final Project Report
Accelerating inference of YOLOv3-tiny on FPGA

[1]Anshuman
Georgia Institute of Technology

[1]Arvind Nataraj Sivasankar
Georgia Institute of Technology

[1]Shreyas Tater
Georgia Institute of Technology

I. Introduction

The object detection technology deals with the problem of detecting instances of objects in images and videos. Applications of this technology can be found in the deployment of advanced intelligent systems like Advanced Driver Assistance Systems (ADAS), medical imaging and video surveillance.

Early works were based on R-CNN [1] which involved two stages, region proposal selecting possible candidates that include an object and a deep neural network responsible for the classification of these regions. As resource sharing between the two parts is limited, the above approach exhibits usually high computational loads and detection latency.

In an attempt to provide object detectors with lower computational requirements and enable object detection in low-power devices, the one-step approach was devised, where the bounding boxes around the objects are predicted directly though the DNN. YOLO [2] is a popular framework in this segment. It relies on a single DNN, DarkNet, in order to predict both the position of the objects as well as their classification. YOLOv3 demonstrates its advantages of both low latency and high classification precision over other competitors [3].

This work addresses the challenging problem of deploying YOLOv3-tiny in a low-power FPGA device with limited resource availability and significant use of the off-chip memory. Performance and resources models are derived that guide the Design Space Exploration (DSE) phase for identifying design points that optimise the latency of the system, meeting at the same time the resource constraints.

II. Problem Description

Many real-time object detection applications today have lower latency as their prime requirement. Most of the algorithms which are used for object detection are resource intensive. For example, R-CNN, Fast R-CNN, Faster R-CNN implement the object detection by two distinct part, region proposal selecting possible candidates that include an object and a deep neural network responsible for the classification of these regions. Since the resource sharing between the two parts is limited, the above approach exhibits high computational loads and detection latency. In order to address the limitations of the two-step approach, a one-step approach, YOLO was introduced.

Even though YOLO has better resource utilization than those of the two-step approaches, it still is resource intensive due to high number of computational layers. To address the computational and latency limitations, tiny-YOLO is introduced. This work addresses the challenging problem of deploying YOLOv3-tiny in a low-power FPGA device with limited resource availability and significant use of the off-chip memory. Performance and resources models are derived that guide the Design Space Exploration (DSE) phase for identifying design points that optimise the latency of the system, meeting at the same time the resource constraints.

III. Existing Work

Several works have been proposed for deploying different versions of YOLO on FPGAs. The authors in [4] implement a parameterised FPGA-tailored architecture for YOLOv3-tiny to optimize latency sensitive applications. Ref. [5] proposed a systolic array architecture, targeted for FPGAs, which uses General Matrix Matrix Multiplication (GEMM) to compute convolution in YOLOv3-Tiny. The work merges batch normalization and convolution processes together, uses quantization to reduce bit precision and adds a new DLQ layer to YOLOv3-Tiny, to maintain similar accuracy with a smaller bit size. Ref. [6] accelerated YOLOv2-Tiny by computing convolutions based on a General Matrix Matrix Multiplication (GeMM) core, designed via OpenCL. Batch normalization and convolution layers are merged in this work as well.

Ref. [7] is related to acceleration of YOLOv3-Tiny and is focused on a high-performance FPGA with much more resources than the low cost FPGA that we aim to target in this work. Ref. [8] is another work which focuses on acceleration of YOLOv2-Tiny, however it requires layer parameters to be stored on-chip. Acceleration of a quantized version of YOLOv2, Sim-YOLOv2, using an FPGA is proposed in Ref. [9], which maximizes throughput of the system. On-chip storage of weights is required in this work and hence it targets a high-performance FPGA, Virtex-7 VC707, which in turn leads to higher power consumption.

IV. Mid-term Status

In line with our Execution Plan, we started working with a Python Code to gain better understanding of the algorithm and have a golden output as defined by the author of the YOLO algorithm. We have used this to develop our own golden C Code to verify the software execution without considering the hardware level features. The C Code encompasses all layers of YOLOv3-tiny: Max Pooling, Upsampling, YOLO, Convolution and Route.

Convolution Block: The block performs direct convolutions of Nin input channels with the corresponding kernels and produces Nout output channels. Internally the block contains three main sub-modules, including an input line buffer,convolution kernels and an output buffer.

Yolo Block: The block implements the functionality of the YOLO layer which is mainly composed of Leaky ReLU activation.

Route Block: This block defines the sequence of the operations involved in the complete algorithm for varying resolution of input images.

Max Pooling Block: This block is responsible for the downsampling operations.

Upsample Block: This block is responsible for the upsampling operations.

The C Code and the HLS code are being implemented in a folder structure similar to the Lab 2 folder hierarchy. For implementing a non-synthesizable golden C code, we define ”sim.cpp” and ”model_conv.cpp” files.

A. *sim.cpp*

This code ensures the following:

1) Reading sample Input Image
2) Reading the trained weights for every convolutional layer
3) Reading the weights for batch normalization operations
4) Type casting to ensure proper data precision
5) Declaring the sequence of layers for YOLO algorithm
6) Ensuring Top-level execution of the individual layer functions and accompanying arguments

B. *model_conv.cpp*

The following operations are defined in this code:

1) Defining a modular convolutional layer with activation function for varying input image sizes
2) Defining the Max Pooling layer operation with varying strides of 1 and 2
3) Defining the Upsampling operation
4) Calculating the probability matrix of the YOLO detection layer

The main challenges that we encountered during this were in creating modular code that can be made Hardware synthesizable with minimal changes. Additionally, since we modified the way the original code reads the weights from the binary file, we had to develop new code to read from multiple binary files instead of reading from a single binary file as was implementing originally. This was necessary to help us in debugging for manually provided input matrices.

V. Incremental update post mid-term

The HLS code for all the 24 layers of the code was implemented. During this implementation of synthesizable code, we realized that fixed point implementation of batch normalization and sigmoid/exponential activation functions would require significant effort and resources. Hence, we skipped their synthesizable code implementations and focused on developing the synthesizable code for the remaining part of the yolov3-tiny network. We re-used tiled convolution/maxpooling modules for different layers of the yolov3-tiny network in order to make sure we use less than 100% of the on-board resources, even if that means starting with a higher latency number. We performed optimizations such as array partitioning, loop pipelining, loop unroll, loop tiling and ping pong buffers.

The FPGA accelerator consists of a three-stage pipeline(Figure 1). The first stage of the pipeline supports the execution of the convolution layer, whose output is accumulated in the second stage of the pipeline. Depending on the network structure that is executed at a given time, the accumulation results are sent for further processing in the Max pooling, Upsample or Yolo layer.

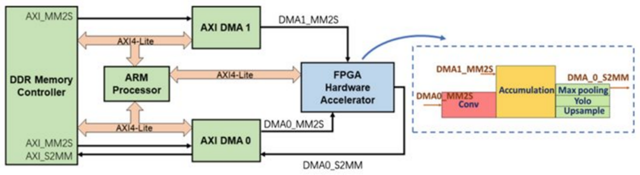
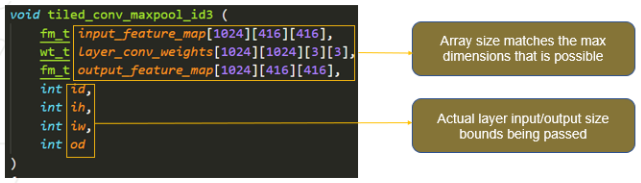


Fig. 1. Overall Accelerator Design and Architecture

Using this approach, we were able to bring down the maximum latency in the HLS synthesis report from 475 seconds(unoptimized) to 40.294 seconds(accelerated) and minimum latency from 3.671 seconds to 0.685 seconds.

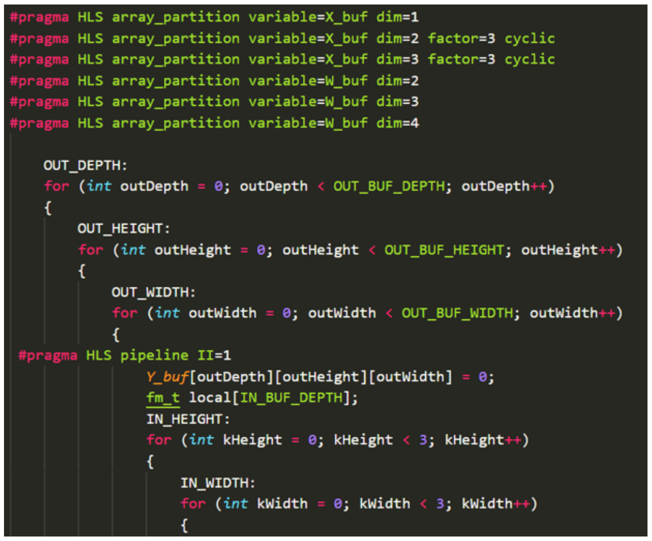


```
void tiled_conv_maxpool_id3 (
    fm_t input_feature_map[1024][416][416],
    wt_t layer_conv_weights[1024][1024][3][3],
    fm_t output_feature_map[1024][416][416],
    int id,
    int ih,
    int iw,
    int od
)
```

Array size matches the max dimensions that is possible
Actual layer input/output size bounds being passed

Fig. 2. Code Snippet: All layer HLS code with reuse of tiled functions

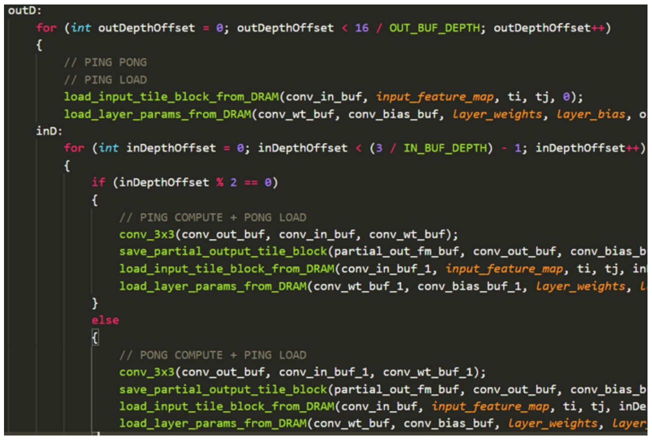
In order to resolve the wide difference between min and max latencies, and accelerate the implementation further, we switched to an alternate approach of using individual tiled convolution/maxpooling modules for the different layers of the yolov3-tiny network. We started with a single layer alone, and performed optimizations on it similar to our earlier approach such as array partitioning, loop pipelining, loop unroll, loop tiling and ping pong buffers, but this time with unique tiled modules instead of sharing them across different layers. We extended the same to 4 layers and achieved a min and max latency of 305 milliseconds.



```
#pragma HLS array_partition variable=X_buf dim=1
#pragma HLS array_partition variable=X_buf dim=2 factor=3 cyclic
#pragma HLS array_partition variable=X_buf dim=3 factor=3 cyclic
#pragma HLS array_partition variable=W_buf dim=2
#pragma HLS array_partition variable=W_buf dim=3
#pragma HLS array_partition variable=W_buf dim=4

    OUT_DEPTH:
    for (int outDepth = 0; outDepth < OUT_BUF_DEPTH; outDepth++)
    {
        OUT_HEIGHT:
        for (int outHeight = 0; outHeight < OUT_BUF_HEIGHT; outHeight++)
        {
            OUT_WIDTH:
            for (int outWidth = 0; outWidth < OUT_BUF_WIDTH; outWidth++)
            {
 #pragma HLS pipeline II=1
                Y_buf[outDepth][outHeight][outWidth] = 0;
                fm_t local[IN_BUF_DEPTH];
                IN_HEIGHT:
                for (int kHeight = 0; kHeight < 3; kHeight++)
                {
                    IN_WIDTH:
                    for (int kWidth = 0; kWidth < 3; kWidth++)
                    {
```

Fig. 3. Code Snippet: Optimization Strategies - Array Partitioning, Loop Pipeline, Loop Unroll

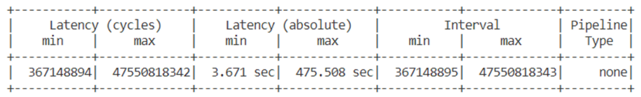


```
outD:
    for (int outDepthOffset = 0; outDepthOffset < 16 / OUT_BUF_DEPTH; outDepthOffset++)
    {
        // PING PONG
        // PING LOAD
        load_input_tile_block_from_DRAM(conv_in_buf, input_feature_map, ti, tj, 0);
        load_layer_params_from_DRAM(conv_wt_buf, conv_bias_buf, layer_weights, layer_bias, o
    inD:
        for (int inDepthOffset = 0; inDepthOffset < (3 / IN_BUF_DEPTH) - 1; inDepthOffset++)
        {
            if (inDepthOffset % 2 == 0)
            {
                // PING COMPUTE + PONG LOAD
                conv_3x3(conv_out_buf, conv_in_buf, conv_wt_buf);
                save_partial_output_tile_block(partial_out_fm_buf, conv_out_buf, conv_bias_b
                load_input_tile_block_from_DRAM(conv_in_buf_1, input_feature_map, ti, tj, in
                load_layer_params_from_DRAM(conv_wt_buf_1, conv_bias_buf_1, layer_weights, l
            }
            else
            {
                // PONG COMPUTE + PING LOAD
                conv_3x3(conv_out_buf, conv_in_buf_1, conv_wt_buf_1);
                save_partial_output_tile_block(partial_out_fm_buf, conv_out_buf, conv_bias_b
                load_input_tile_block_from_DRAM(conv_in_buf, input_feature_map, ti, tj, inDe
                load_layer_params_from_DRAM(conv_wt_buf, conv_bias_buf, layer_weights, layer
```

Fig. 4. Code Snippet: Optimization Strategies - Loop Tiling, Ping Pong Buffer

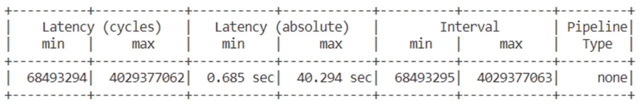
VI. Results

Latency snapshots of unoptimized and optimized synthesizable code for all 24 layers(without batch normalization, sigmoid/exponential activation functions)are shown below:



```
+----------+-----------+----------+-----------+----------+-----------+---------+
|    Latency (cycles)  |  Latency (absolute)  |       Interval       | Pipeline|
|    min   |    max    |    min   |    max    |    min   |    max    |   Type  |
+----------+-----------+----------+-----------+----------+-----------+---------+
| 367148894| 47550818342| 3.671 sec| 475.508 sec| 367148895| 47550818343|     none|
+----------+-----------+----------+-----------+----------+-----------+---------+
```

Fig. 5. Latency: all layer unoptimized HLS code



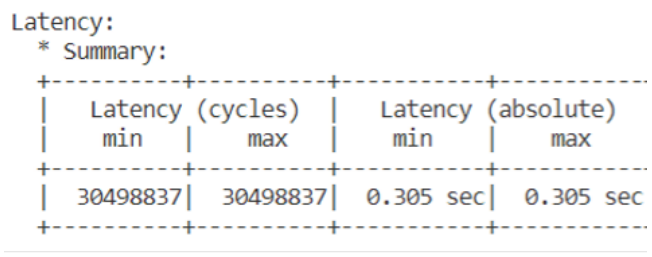
```
+----------+-----------+----------+-----------+----------+-----------+---------+
|    Latency (cycles)  |  Latency (absolute)  |       Interval       | Pipeline|
|    min   |    max    |    min   |    max    |    min   |    max    |   Type  |
+----------+-----------+----------+-----------+----------+-----------+---------+
|  68493294| 4029377062| 0.685 sec| 40.294 sec|  68493295| 4029377063|     none|
+----------+-----------+----------+-----------+----------+-----------+---------+
```

Fig. 6. Latency: all layer optimized HLS code

Result of optimizing a single layer alone is shown in table below.

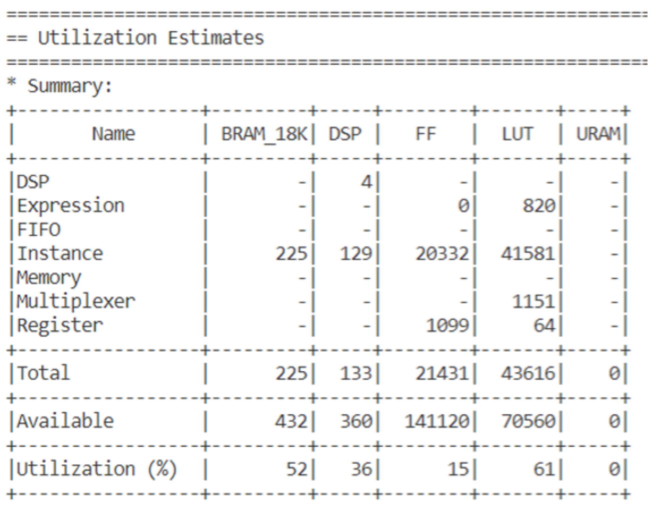
	Latency (sec)	Speedup
Unoptimized	3.2	1
Ping Pong Buffer	2.02	1.584158416
Loop Unroll + Pipelining	0.186	17.20430108
Array Partitioning	0.0549	58.28779599

Results of extending single layer optimizations to 4 layers(with independent tiled convolution/maxpooling modules) are shown in figures 7 and 8.



```
Latency:
  * Summary:
  +---------+---------+----------+----------
  |  Latency (cycles) |  Latency (absolute)
  |   min   |   max   |    min   |    max
  +---------+---------+----------+----------
  | 30498837| 30498837| 0.305 sec| 0.305 sec
  +---------+---------+----------+----------
```

Fig. 7. Latency: 4 layer optimized HLS code



```
================================================================
== Utilization Estimates
================================================================
* Summary:
+-----------------+---------+-----+--------+-------+-----+
|       Name      | BRAM_18K| DSP |   FF   |  LUT  | URAM|
+-----------------+---------+-----+--------+-------+-----+
|DSP              |        -|    4|       -|      -|    -|
|Expression       |        -|    -|       0|    820|    -|
|FIFO             |        -|    -|       -|      -|    -|
|Instance         |      225|  129|   20332|  41581|    -|
|Memory           |        -|    -|       -|      -|    -|
|Multiplexer      |        -|    -|       -|   1151|    -|
|Register         |        -|    -|    1099|     64|    -|
+-----------------+---------+-----+--------+-------+-----+
|Total            |      225|  133|   21431|  43616|    0|
+-----------------+---------+-----+--------+-------+-----+
|Available        |      432|  360|  141120|  70560|    0|
+-----------------+---------+-----+--------+-------+-----+
|Utilization (%)  |       52|   36|      15|     61|    0|
+-----------------+---------+-----+--------+-------+-----+
```

Fig. 8. Resource Utilization: 4 layer optimized HLS code

VII. Future Work

We could extend our 4-layer implementation with independent tiled convolution/maxpooling modules to all 24 layers or maximum number of layers possible, so that almost 100% of on-board FPGA resources are utilized. We could also do a literature survey of fixed-point implementation of batch normalization, sigmoid/exponential activation functions on a low end FPGA in order to fully implement the original yolov3-tiny network structure.

References

[1] R. Girshick, J. Donahue, T. Darrell and J. Malik, "Rich Feature Hierarchies for Accurate Object Detection and Semantic Segmentation," 2014 IEEE Conference on Computer Vision and Pattern Recognition, 2014, pp. 580-587, doi: 10.1109/CVPR.2014.81.

[2] J. Redmon, S. Divvala, R. Girshick and A. Farhadi, ”You Only Look Once: Unified, Real-Time Object Detection,” 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 2016, pp. 779-788, doi: 10.1109/CVPR.2016.91.

[3] Redmon, J., Farhadi, A.: YOLOv3: an incremental improvement, April 2018. https://pjreddie.com/media/files/papers/YOLOv3.pdf

[4] Yu, Z., Bouganis, CS. (2020). A Parameterisable FPGA-Tailored Architecture for YOLOv3-Tiny. In: Rincón, 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.

[5] 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.

[6] Wai, Y.J., bin Mohd Yussof, Z., bin Salim, S.I., Chuan, L.K.: Fixed point implementation of Tiny-Yolo-v2 using OpenCL on FPGA. Int. J. Adv. Comput. Sci. Appl. 9(10), 506–512 (2018)

[7] 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.

[8] Wei, G., Hou, Y., Cui, Q., Deng, G., Tao, X., Yao, Y.: YOLO acceleration using FPGA architecture. In: 2018 IEEE/CIC International Conference on Communications in China (ICCC), pp. 734–735, August 2018

[9] Nguyen, D.T., Nguyen, T.N., Kim, H.: A high-throughput and power-efficient FPGA implementation of YOLO CNN for object detection. IEEE Trans. Very Large Scale Integr. (VLSI) Syst. 27(8), 1861–1873 (2019)