Proceeding of the IEEE
International Conference on Robotics and Biomimetics
Dali, China, December 2019

# FPGA-based Power Efficient Face Detection for Mobile Robots

Cong Fu*
*Department of Mechanical Engineering*
*University of Michigan, Ann Arbor*
*Ann Arbor, Michigan, USA*
*ffucong@gmail.com*

Yunxuan Yu*
*Department of Electrical and Computer Engineering*
*University of California, Los Angeles*
*Los Angeles, California, USA*
*yunxuan.yu@hotmail.com*

***Abstract*— Autonomous mobile robots need perception module to understand nearby environments, avoid obstacles and, most importantly, interact with the humans in the dynamic environments. In order to perform human robot interaction for mobile robots, face detection is an inevitable and important function. In recent years, deep learning has emerged to be the most accurate approach for face detection problems. However, deep learning models have very high computational density, therefore cannot be executed in real-time on the embedded low-end processors that most of the robot systems equip with. Switching to high-end CPU or GPU generally means much higher cost and power consumption. Meanwhile, FPGA is well suited for robot application because it provides high computing power with low cost and high power efficiency. In this paper we design an FPGA based system for the acceleration of MTCNN, which is one of the most accurate face detection neural networks. We evaluate the system using a Zynq 706 board and compare the performance with embedded CPU and popular GPU edge computing platform Jetson TX2. Our FPGA system has $40\times$ lower latency than Jetson TX2 with $2.5\times$ higher power efficiency, making it a promising candidate for egde computing on mobile robot applications.**

***Index Terms*— FPGA, CNN, mobile robot**

## I. Introduction

Service robots are increasingly employed in various indoor environments. Semantic understanding of the world and situation awareness are crucial for the service robot to safely interact with the dynamic environments and obstacles, especially humans. When it comes to interact with humans during navigation, face detection is an important module that enables robots to locate and identify humans and even understand the intention of humans. For example, guiding robots need to track and identify the person to be guided and meanwhile avoid other humans to accomplish navigation task [1]. Moreover, education robot needs to detect children's face to analysis their emotion in order to reason about their psychological activity. And medical care robot needs to track patients' gaze or face orientation in case that they needs help.

Recent years, deep learning has emerged rapidly and shown superior performance than traditional computer vision algorithms in many aspects [2]. Face detection area also utilizes deep learning to achieve remarkable performance [3]. MTCNN [4] is the state of the art deep CNN, which can jointly conduct both face detection and face alignment task in a unified framework. However, deep learning methods require large amount of computing operations. For example, VGG [5] net has 30 billion FLOPs and Openpose [6] has 300 billion FLOPs. The high computation requirements put high demand on the processing capability of robot processors, and most embedded processors on the service robot cannot run these networks in real-time. One substitute is high-performance processors(GPU), which is power consuming and expensive for service robot. Moreover, GPU processes data in batch for high resource utilization to achieve high throughput, which leads to high latency. Meanwhile, robot needs to handle sequential data perceived from the environments in real-time with low latency. FPGA is able to provide high performance with low power consumption and low latency, making it suitable for deep learning acceleration on power limited robot system with real-time requirements [7]–[9].

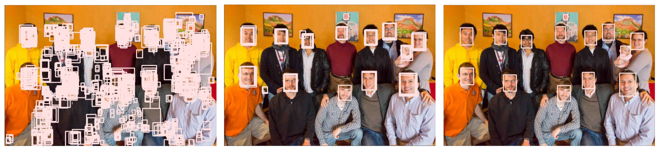
(a) Pnet Result (b) Rnet Result (c) Onet Result

Fig. 1: Output of three networks from MTCNN. P-Net proposes bounding boxes for potential faces. R-Net refines the result of P-Net. And O-Net gives the final bounding boxes and facial landmarks.

In this paper, we propose an FPGA-based MTCNN accelerator system to accelerate face detection task. We use Zynq to implement our accelerator. It is equipped with ARM processor, therefore it is also capable to collect and process data from other sensors for the robot. Our FPGA design highlights the controllable layer instruction, which can accelerate three separate networks in MTCNN using a uniform computation engine to maximum run-time resource utilization. The computation engine explores multiple level

*Both authors contributed equally to this manuscript

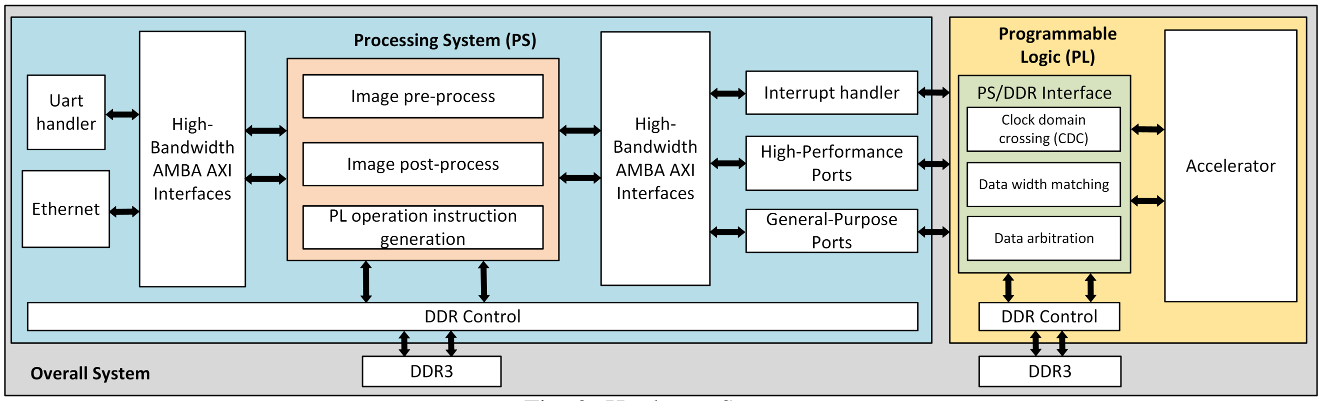

Fig. 2: Hardware System

of parallelisms, which can be adjusted for different networks based on their characteristics for high throughput. The major contributions of this paper are summarized as follows:

- To the best of our knowledge, we are the first to propose FPGA-based accelerator for the popular cascaded face detection network MTCNN.
- We test the accelerator on a set of images with high resolution and multiple faces to match the actual application scenario. Face detection latency using MTCNN on our accelerator is $2.73\times$ faster than CPU and $40\times$ faster than GPU on Jetson TX2, which can perform real-time face detection on service robot.
- Power efficiency of running MTCNN is $2.5\times$ higher than GPU on Jetson TX2, which is well suited for service robot application.

The rest of the paper is structured as follows. In Section II we introduce the background of MTCNN. In Section III we present the implementation of the hardware acceleration system. And in Section IV we evaluate the performance of our system and compare with CPU and GPU baselines. Section V concludes the paper.

## II. Background

Face detection is important to robot application when robots need to interact with humans in social environments. Multi-task Cascaded Convolutional Network (MTCNN) is a unified cascaded CNNs based framework that integrates both face detection and face alignment tasks. MTCNN consists of three cascaded networks, Proposal Network(P-Net), Refine Network(R-Net) and Output Network(O-Net). In the first stage, P-Net generate proposal bounding boxes through a shallow CNN. And then R-Net will refine the result of the P-Net to decrease the numbers of potential bounding boxes and adjust its coordinates. Finally, O-Net takes the bounding boxes from R-Net as input and generate the final bounding boxes and five facial landmarks.

Next we introduce more details of this network. P-Net can take in an image with any size. First, the original image will be resized to different scales to build an image pyramid,

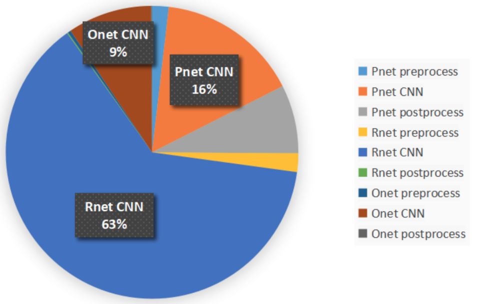

Fig. 3: Averaged Profiling results on a subset of the WIDER FACE dataset [10]

which is the input of P-Net. P-Net will generate the candidate bounding boxes on each scaled image, and then employ non-maximum suppression (NMS) to reduce overlapped bounding boxes. After first stage, P-Net will generate $m$ candidate bounding boxes. Next, R-Net will take in the P-Net bounding boxes, and resize them into $24 \times 24$. R-Net then refine the result of P-Net to reduce a lot of false detected candidates. After R-Net performing NMS, only $n$ bounding boxes will be fed into the O-Net. O-Net finally takes the image size of 48x48, and output the final bounding boxes and five facial landmarks.

## III. Implementation

### A. System overview

As shown in Fig. 2, our acceleration system includes three main parts, i.e., (1). FPGA programming logic (PL); (2). Embedded ARM processor core (PS); (3). On-board memory units including two DDR3 connecting to PL and PS, respectively. The PS and PL are connected through high speed AXI HP ports with bandwidth up to $256bits$ at 250 MHZ, which guarantees fast data communication between PS and PL.

In order to decide task distribution on PL and PS, we perform detailed profiling for each step of the MTCNN computation, which is shown in Fig. 3. The *CNN* part of *Pnet*, *Rnet* and *Onet* takes 63%, 16% and 9% of the overall

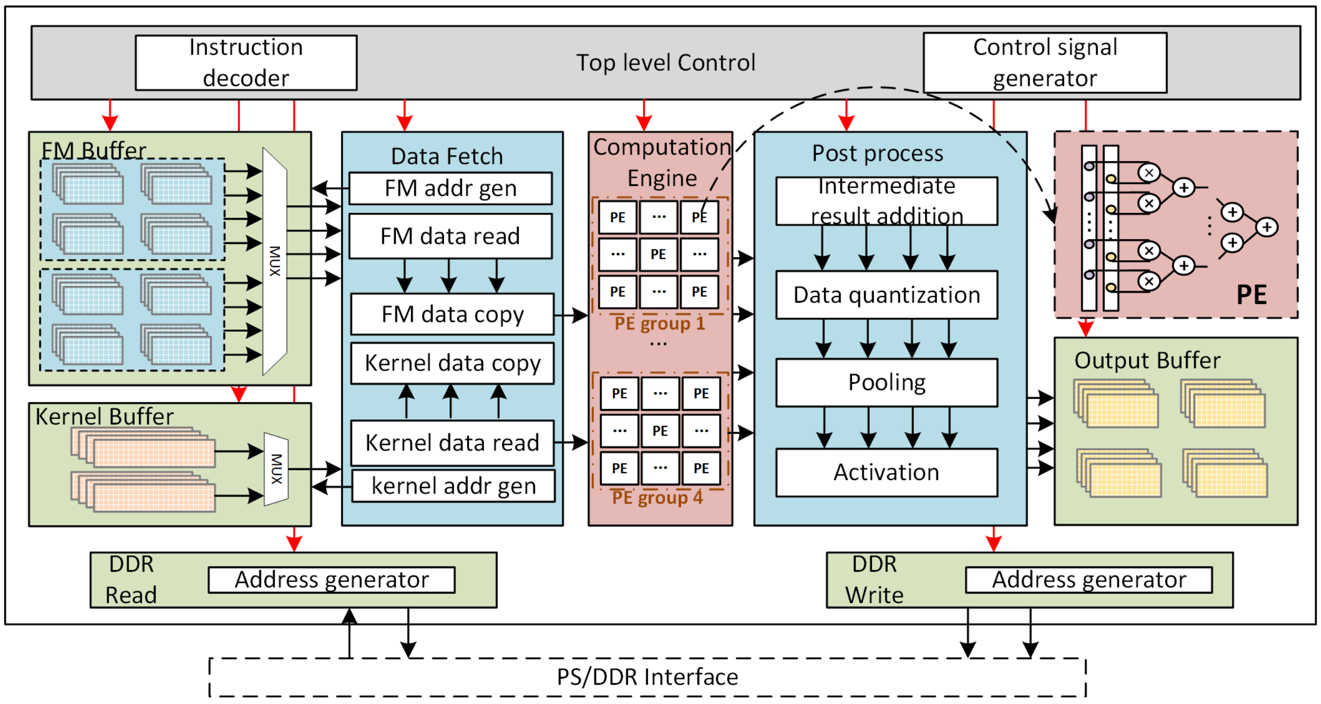

Fig. 4: FPGA accelerator architecture

inference time, which adds up to 88%. While the pre-/post-process operations only consumes 12% of the overall time. Therefore, we decided to deploy the *CNN* computation to FPGA (PL) and rest of the steps to PS. As shown in Fig. 2, the PS system accepts input images from Ethernet connected camera, then handle the pre-/post-process of each network. Whenever the computation of *CNN* is required, the *PL Layer instruction generator* will send instruction out to PL for *CNN* execution. In the end, a list containing all coordinates of faces in the robot view will be send out to master-computer(ROS) through UART port.

## B. PL parallelism Analysis

In this section we analyze the explorable parallelism hidden within the realtime computation process of *CNN* part, which can be utilized for FPGA acceleration.

***FM Batch*** runs multiple input FMs in parallel. As introduced in section II, *Rnet* and *Onet* are called multiple times with different bounding box inputs. Different calling can be calculated in parallel as no data dependency exists among them. ***FM Batch*** helps fully utilize the on-chip computation resources.

***Layer Pipeline*** implements the complete network dataflow on-chip, with each layer having its own acceleration unit. The computation of different layers are running in a pipelined fashion to increase the overall throughput. ***Layer Pipeline*** implies the hidden assumption that all layers are called the same number of times, otherwise part of the computation resource will be idle. However, as a cascaded network, MTCNN calls *Rnet* and *Onet* based on the results of *Pnet*. The calling times of *Pnet*, *Rnet* and *Onet* can differ a lot. This runtime-dynamic network calling schedule renders ***Layer Pipeline*** inefficient for MTCNN.

***Feature Map (FM) Parallelism*** computes a window of output FM pixels in parallel, which boosts on-chip FM data reuse and alleviates bandwidth bottleneck [11]. The window size is chosen to be the common factor of layer sizes for resource efficiency maximization. However, the input size of *Pnet* relies entirely on the input image size and the scaling factors. Moreover, part of the layers from *Onet* and *Rnet* have input sizes of $4 \times 4$ or smaller. The unpredictable *Pnet* and small *Onet*/*Rnet* layer input sizes make it difficult to find a suitable window size for large number of parallelism.

***Kernel Parallelism*** expands all the computation within a kernel window for parallelization, which normally requires the number of computation units to be the multiple of expanded kernel size. Moreover, extra data rearrangement modules specific to kernel size is needed for data expansion. MTCNN contains various kernel sizes of $3\times3$, $2\times2$ and $1\times1$, which requires multiple extra data rearrangement modules with excessive resource consumption.

***Channel Parallelism*** computes different input and output channels in parallel. We observe that the channel numbers of MTCNN are multiples of 16 in most cases, i.e., 16, 32, 64, 128 and 256. The uniform values of channel number enables the development of a uniform accelerator, making ***Channel Parallelism*** an ideal option.

In summary, ***FM Batch*** and ***Channel Parallelism*** are optimal choices of parallelism with regard to the acceleration of MTCNN.

### C. *Micro-architecture of PL accelerator*

The micro-architecture of the CNN accelerator implemented in the PL side is shown in Fig. 4.

It is composed of four parts, i.e., (1). *Computation Engine* for computation intensive operations; (2). *Memory system* that handles the on-chip data caching for fast data access; (3). *Data process* for data fetch, data rearrangement, data quantization as well as pooling and activation; (4). *Control system* for instruction decoding and control signal generation.

*1) Memory System:* The *Memory system* includes on-chip *BRAM buffer groups* and *data access control modules* for off-chip data access and caching. As shown in Fig. 4, there are three groups of on-chip *BRAM buffers*. *FM buffers* are utilized to store input FM data fetched from off-chip memory. Each *FM buffers* contains up to four banks to provide sufficient bandwidth for ***FM Batch*** choices and each bank has a bandwidth of $16bits \times 16$. *Kernel buffer* caches kernel weights that will be reused throughout the current computation round (As discussed in section III-C.2). We employ Ping-Pong structure for both *FM buffer* and *Kernel buffer* to hide the memory access latency. Specifically, two sets of identical buffers are implemented, when using the data from buffer a, buffer b is being loaded at the same time. Then the role of two buffers will be switched for the next round. In this way, loading latency is covered by the computation latency. We also implement an *output buffer* for intermediate result storage. The *Output buffer* has the same number of banks as *FM buffer* for ***FM Batch*** size matching.

For the *data access control modules*, we implement *DDR read* and *DDR write*. *DDR read* is designed to fetch data from off-chip memory DDR3 given starting address, read number and data type, since we set different read pattern for *FM data* and *Kernel weights data*. *DDR Write* sends the output *FM data* back to designated address of DDR3. An *address generator* is included for each *data access control module* to produce new DDR address at each clock cycle. It should be noted that DDR3 has a periodically refreshing schedule, during which read and write are forbidden. Therefore we also employ the handshake mechanism in both *DDR read* and *DDR write* to ensure we only read or write data when DDR3 is available, to avoid loss of data.

*2) Computation Engine (CE):* The architecture of *CE* is designed to explore ***FM Batch*** and ***Channel Parallelism***. As shown in Fig. 4, we compose the basic processing element (*PE*) structure with one multiplier array followed by an adder tree. The number of multipliers within one PE is set to be 16 according to the channel property of MTCNN. A total of 64 *PE*s are implemented, which are decomposed evenly into 4 groups for up to 4 ***FM Batch***. The computation pattern we employ can be described in Fig. 4. Each group accept FM input from one *FM banks*, and is able to compute 16 input channel and 16 output channels. The four *FM banks* can be utilized to store different input FM or different slice from

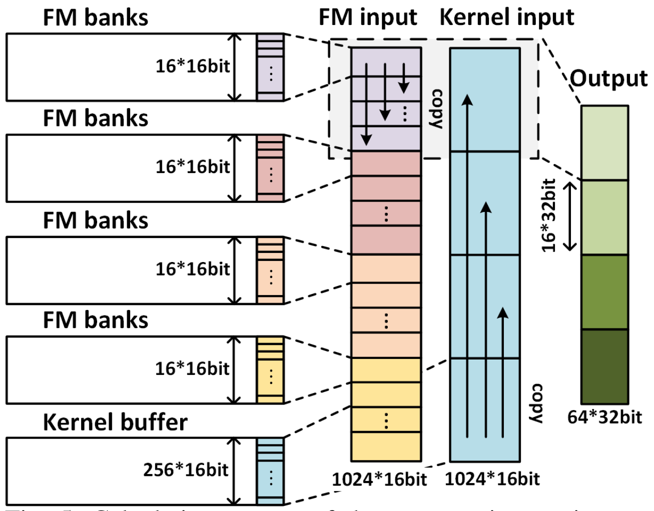

Fig. 5: Calculation pattern of the computation engine. *FM input* and *Kernel input* represent the two input vectors to the PE groups. Data from Kernel buffer is copied 4 times for different output pixels within same FM or different output FMs. Data from FM banks are copied 16 times for 16 output channels.

one input FM, based on the ***FM batch*** setting. For example, for one layer with input FM size $25 \times 25$, decomposing the input FM into 4 slices poses waste on the *FM banks* resource, which have depth 1024. Therefore, we load 4 different inputs and choose ***FM Batch*** = 4. For one layer with input size $224 \times 224$ and is not suitable for ***FM Batch***, choosing ***FM Batch*** = 1 and decomposing the input into 4 *FM banks* for multiple rounds is more suitable.

*3) Data Process: Data Process* includes a series of modules handling the on-chip data access and other non-computational-intensive data operations.

- *Data Fetch (DF)* reads FM and kernel weights data from on-chip buffers. It also conducts rearrangement before sending the data to *CE*. For example, FM data will be copied 16 times for different output channels and Kernel weights data will be copied 4 times, as kernel weights are shared among different input images.
- *Intermediate result addition (IRA)* module takes calculation results stream from *computation engine*, generates addresses and write data to different *output buffers* for temporary storage. When a new round of calculation results come in, IRA will fetch corresponding partial result from the *output buffer* and conducts addition with the new data.
- *Data Quantization (DQ)* performs shift, cut and saturation for results after fixed-point arithmetic. We employ $16bit$ fixed-point for both input FM data and kernel weights data, then the output FM will be quantized back to $16bit$. Previous researches have verified that the

influence on model accuracy with regard to data-width reduction to $16bit$ is negligible [12] [13].

- *Pooling* module performs maximum pooling operation. There are two configurations of pooling exist in MTCNN, $3 \times 3$ with $2 \times 2$ strides and $2 \times 2$ with $2 \times 2$ strides. We use a typical cascaded comparator architecture for both pooling configurations.
- *activation* module performs *PReLu* operations, which is represented as:

$$f(x) = \begin{cases} x \ \ if \ x > 0 \\ \alpha x \ otherwise, \end{cases} \tag{1}$$

where each output channel has its own $\alpha$ value.

*4) Control System:* The *Control System* decodes the *layer instruction* from PS and generates control signals for the rest of the modules. Each *layer instruction* has $185bits$, which contains the detailed configuration of one convolutional/fully connected layer, along with its corresponding pooling layer and activation layer. A *layer done* signal will be sent back to PS and trigger the transmission of next *layer instruction*. Table I shows the details of the *layer instructions* definition. After decoding the *layer instruction*, *Control system* generates start signals and computes parameters for other modules at each computation round, based on runtime program status. Algorithm 1 illustrates the operating principle of the *Control system*.

TABLE I: Layer Instruction Definition

| Bit | Parameter | Parameter definition |
|---|---|---|
| [0:0] | Layer type | Convolutional layer or fully connected layer |
| [25:1] | FM Addr | DDR starting address for input FM and Kernel weights |
| [50:26] | Ker Addr | |
| [75:51] | Out FM Addr | DDR destination address for output FM |
| [91:76] | Ker Config | Kernel size and stride |
| [131:92] | FM Config | Input FM size and output FM size |
| [147:132] | Pooling Config | Pooling size and stride |
| [167:148] | Channel Config | Input channel and Output channel number |
| [170:168] | Batch | ***FM Batch*** number |
| [184:171] | Block size | If the input layer size is larger than on-chip input FM buffer, *Block size* defines the block input FM size after decomposition |

**Algorithm 1** Operating process of *Control system*

```
1:  function CONTROL SYSTEM($Layer\ Instruction$)
2:      Layer instruction decoding
3:      for $bx \leftarrow 1, \left\lceil \frac{fm_w}{block_w} \right\rceil$ do
4:          for $bx \leftarrow 1, \left\lceil \frac{fm_w}{block_w} \right\rceil$ do
5:              for $chout \leftarrow 1, \left\lceil \frac{Ch_{out}}{Ch_{out\_slice}} \right\rceil$ do
6:                  for $chin \leftarrow 1, \left\lceil \frac{Ch_{in}}{Ch_{in\_slice}} \right\rceil$ do
7:                      DDR read address computation
8:                      Send start signal to DDR read
9:                      for $kky \leftarrow 1, K_y$ do
10:                         for $kkx \leftarrow 1, K_x$ do
11:                             Send start signal to DF
12:                             Send start signal to CE
13:                             Send start signal to IRA and DQ
14:                         end for
15:                     end for
16:                 end for
17:                 Send start signal to Pooling and Activation
18:                 DDR write address computation
19:                 Send start signal to DDR write
20:             end for
21:         end for
22:     end for
23:     Send Layer done signal to PS
24: end function
```

## D. PS/DDR Interface

We implement an interface module to handle the data communication among PL accelerator, DDR3 and PS. Frequency domain crossing and data width matching are involved using a group of FIFOs. There are four available data paths provided by the interface module, i.e., (1). PS data read and write to PL DDR3; (2). Accelerator data read and write to PL DDR3; (3). PS data write to accelerator; (4). Accelerator data write to PS DDR3.

## E. Functionality of PS

In this section, we explain the role of PS in our *Hardware acceleration system.* PS handles the overall inference process of MTCNN, and calls PL for CNN acceleration when necessary. The functionalities of PS are listed as follows:

- System initialization. PS will write the complete kernel weights of three networks to the designated address on PL DDR3. Meanwhile, PS stores the complete *layer instruction sequence (LIS)*s for computing the MTCNN, as well as Hyper-parameters such as *Pnet* scale and $size_{min}$. All the aforementioned data are passed to PS through *Uart Handler* from the PC. The system initialization is done only once.
- Image input. When the system is started, PS will receive images captured by the camera sensor from Ethernet port.
- The overall inference, which is explained in detail in Algorithm 2. The image data transmission from PS to PL DDR3 is completed through the HP AXI port. The PL returned *Layer done* signal is sent to the PS interrupt handler.

## IV. Experiment

## A. Experiment Setting

We use an Xilinx Zynq-7000 XC7Z045 SoCs device (on ZC706 board) with two ARM® Cortex$^{TM}$-A9 MPCores for PS and one Kintex7 FPGA for PL, for the system implementation and evaluation. The FPGA accelerator is running at 200 MHZ. The resource utilization of FPGA is listed in Table. II. A total of 64 PEs with $16 \times 64 = 1024$ $16bit \times 16bit$ multipliers are implemented. We use both DSP and LUT to instantiate multipliers.

We also employ edge computing CPU and GPU for performance comparison. For CPU baseline, we use an Intel(R) Core(TM) i7-8700K CPU. For GPU baseline, we employ the popular Jetson TX2 with 256 CUDA cores. A

**Algorithm 2** MTCNN inference process on PS

```
1:  function INFERENCE($Input\ image$)
2:      Get input image size $w \times h$
3:      $w_{cur} = w,\ h_{cur} = h$
4:      while $min(w_{cur}, h_{cur}) > size_{min}$ do
5:          for $i \leftarrow 1, size(Pnet\_LIS)$ do
6:              Pnet preprocess
7:              Transmit input image to PL DDR3
8:              Send $Pnet\_LIS[i]$ to PL with start signal
9:              Wait for Layer done signal that PL returned
10:         end for
11:         Wait for accelerator to write results back to PS DDR
12:         Pnet postprocess
13:         $w_{cur}* = scale,\ h_{cur}* = scale$
14:     end while
15:     Pnet overall NMS, find bounding box number $n$
16:     Bounding box correction
17:     for $i \leftarrow 1, n$ do
18:         Rnet preprocess
19:         Transmit input bounding box to PL DDR3
20:         for $i \leftarrow 1, size(Rnet\_LIS)$ do
21:             Send $Rnet\_LIS[i]$ to PL with start signal
22:             Wait for Layer done signal that PL returned
23:         end for
24:         Wait for accelerator to write results back to PS DDR
25:     end for
26:     Rnet overall NMS, find bounding box number $m$
27:     Bounding box calibration
28:     for $i \leftarrow 1, n$ do
29:         Onet preprocess
30:         Transmit input bounding box to PL DDR3
31:         for $i \leftarrow 1, size(Onet\_LIS)$ do
32:             Send $Onet\_LIS[i]$ to PL with start signal
33:             Wait for Layer done signal that PL returned
34:         end for
35:         Wait for accelerator to write results back to PS DDR
36:     end for
37:     Onet NMS, find final bounding box number $f$
38:     Bounding box calibration
39: end function
```

PN2000 electricity usage monitor is utilized for board power measurement.

The runtime of MTCNN is heavily dependent on the size and the potential number of faces in the input image, which decide the number of times *Pnet*, *Rnet* and *Rnet* are called. For example, input image A with size $1024 \times 681$ and 13 faces takes $374.5ms$ on CPU, while input image B with size $224 \times 224$ with single face takes only $32.7ms$. Therefore, based on the application scenario of service robots, we compile a testset containing 200 images of high resolution and multiple faces from WIDER FACE dataset, for the speed evaluation of MTCNN in complex crowded scenes.

TABLE II: FPGA (PL) Resource Utilization.

| LUT | FF | BRAM | DSP |
|---|---|---|---|
| 133783(61.21%) | 222456(50.88%) | 196(35.96%) | 880(97.77%) |

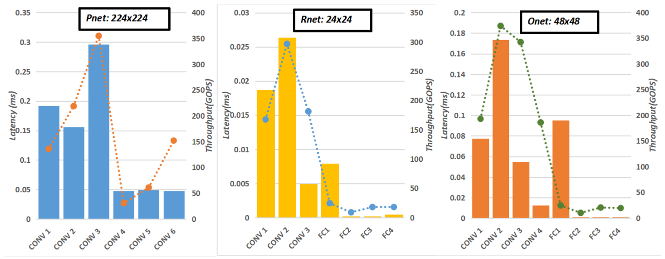

Fig. 6: Latency and throughput evaluation for *Pnet*, *Rnet* and *Onet*. The bars represent latency values, which corresponds to left axis. The line depicts throughput, with value labeled on right axis. (1). *Pnet* is running at ***FM Batch*** = 1 mode, and the overall latency for processing one $224 \times 224 \times 3$ input image takes $0.788ms$; (2). *Rnet* is running at ***FM Batch*** = 4 mode with 4 $24 \times 24 \times 3$ input bounding box. The latency for computing 4 inputs is $0.058ms$; (3) *Onet* is also running at ***FM Batch*** = 4 mode with 4 $48 \times 48 \times 3$ inputs, which takes $0.416ms$.

## B. Experiment result

Fig. 6 shows the latency and throughput evaluation for each layer of *Pnet*, *Rnet* and *Onet*. It should be noted that *Onet* and *Rnet* are running under ***FM Batch*** $= 4$ to fully utilize the computation resources. Therefore, the time displayed in the graph corresponds to the latency of accelerating 4 same layers at the same time. For *Pnet*, although called multiple times, its input sizes are different for each calling, thereby using ***FM Batch*** $= 1$ is more suitable. It can be seen that *conv2* or *conv3* from all three networks usually has the highest throughput, from 296.65 GOPS to 355.00 GOPS. It is because the network starts to expand the channel dimension at these layers, which increase potential parallelism and leads to high computation resource utilization. Meanwhile, large channel number accompanied with large input FM size (close to input image) dramatically increases the computation requirement, making *conv2* or *conv3* with the longest latency.

The platform comparison results are listed in Table. III, where we evaluate the performance of three platforms from several aspects. The *Averaged Latency* is a crucial criterion for real-time performance evaluation, as it indicates the time gap between sending the image to the system and getting the detected output. Meanwhile, FPS represents the generally processing capability of the system. A system running on high batch size, say 32, may have a high FPS, however it may take several seconds to get the 32 outputs all together after feeding the inputs in, which seriously delays the response of robots. We also computes the *Power per frame* to evaluate the power efficiency of different platforms.

The *Input Batch size* indicates the number of input images we computing in parallel. The *Batch size* row list out the number of inputs we utilized for each network. For edge GPU we employ two *Input Batch size* settings, the first computes one input image at a time as our acceleration system for

TABLE III: Comparison with GPU and CPU baselines.

| Platform | Intel(R) Core(TM) i7-8700K CPU | Jetson TX2 | | ZC706 board |
|---|---|---|---|---|
| Frequency | 3.70GHZ | ARM: 1.74GHZ GPU:998MHz | | ARM:1GHZ FPGA:200MHZ |
| Input Batch size | Input image batch = 1 | Input image batch = 1 | Input image batch = 32 | Input image batch = 1 |
| Network Batch size | All networks = 1 | *Pnet* = 1, *Rnet* = *Onet* = 128 | *Pnet* = 32, *Rnet* = *Onet* = 128 | *Pnet* = 1, *Rnet* = *Onet* = 4 |
| Averaged latency (ms)[a] | 230 | 3367 | 7778 | **84** |
| FPS | 4.16 | 0.30 | 4.32 | **11.9** |
| Power/frame (J) | 14.42 | 13.33 | 1.85 | **0.75** |

a: The latency represents the time period between sending the image in and getting the detected faces. Batch mode helps the overall throughput, but has negative influence on the latency. The value reported here are averaged over 200 images.

latency evaluation. For the second one we utilize GPU for its full capacity with maximum *Batch size* for FPS comparison. It can be seen that for the *Latency* evaluation, our FPGA-based acceleration system is $40\times$ faster than the Jetson Tx2, since the limited input image batch size cannot bring GPU to its full computation capacity. For the FPS comparison, our system performs $11.9/4.32 = 2.75\times$ better than GPU with full batch size, which is benefited from our specific designed acceleration architecture targeted on MTCNN. Moreover, FPGA exhibits $2.5\times$ higher power efficiency compared with GPU, making it a suitable platform for battery limited robot applications.

## V. Conclusion

In this paper, we propose an FPGA-based acceleration system for fast and power efficient face detection task for mobile robots. We include a customized accelerator implemented on FPGA for *CNN* calculation and a Processing System for data communication and image processing. Our accelerator system achieves $40\times$ lower latency than Jetson TX2 with $2.5\times$ higher power efficiency, exhibiting the good potential to be employed on general mobile service robots.

## References

[1] T. Kruse, A. K. Pandey, R. Alami, and A. Kirsch, "Human-aware robot navigation: A survey," *Robot. Auton. Syst.*, vol. 61, pp. 1726–1743, Dec. 2013.

[2] J. Walsh, N. O' Mahony, S. Campbell, A. Carvalho, L. Krpalkova, G. Velasco-Hernandez, S. Harapanahalli, and D. Riordan, "Deep learning vs. traditional computer vision," 04 2019.

[8] Y. Yu, C. Wu, T. Zhao, K. Wang, and L. He, "Opu: An fpga-based overlay processor for convolutional neural networks," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 2019.

[3] S. Balaban, "Deep learning and face recognition: the state of the art," *CoRR*, vol. abs/1902.03524, 2019.

[4] K. Zhang, Z. Zhang, Z. Li, and Y. Qiao, "Joint face detection and alignment using multi-task cascaded convolutional networks," *CoRR*, vol. abs/1604.02878, 2016.

[5] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," in *International Conference on Learning Representations*, 2015.

[6] Z. Cao, G. Hidalgo, T. Simon, S.-E. Wei, and Y. Sheikh, "OpenPose: realtime multi-person 2D pose estimation using Part Affinity Fields," in *arXiv preprint arXiv:1812.08008*, 2018.

[7] X. Zhang, J. Wang, C. Zhu, Y. Lin, J. Xiong, W.-m. Hwu, and D. Chen, "Dnnbuilder: an automated tool for building high-performance dnn hardware accelerators for fpgas," in *Proceedings of the International Conference on Computer-Aided Design*, p. 56, ACM, 2018.

[9] P. G. Mousouliotis, K. L. Panayiotou, E. G. Tsardoulias, L. P. Petrou, and A. L. Symeonidis, "Expanding a robot's life: Low power object recognition via fpga-based dcnn deployment," in *2018 7th International Conference on Modern Circuits and Systems Technologies (MOCAST)*, pp. 1–4, IEEE, 2018.

[10] S. Yang, P. Luo, C. C. Loy, and X. Tang, "Wider face: A face detection benchmark," in *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016.

[11] Z. Du, R. Fasthuber, T. Chen, P. Ienne, L. Li, T. Luo, X. Feng, Y. Chen, and O. Temam, "Shidiannao: Shifting vision processing closer to the sensor," in *ACM SIGARCH Computer Architecture News*, vol. 43, pp. 92–104, ACM, 2015.

[12] J. Qiu, J. Wang, S. Yao, K. Guo, B. Li, E. Zhou, J. Yu, T. Tang, N. Xu, S. Song, *et al.*, "Going deeper with embedded fpga platform for convolutional neural network," in *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pp. 26–35, ACM, 2016.

[13] J. H. Lee, S. Ha, S. Choi, W.-J. Lee, and S. Lee, "Quantization for rapid deployment of deep neural networks," *arXiv preprint arXiv:1810.05488*, 2018.