IEEE Access
Multidisciplinary Rapid Review Open Access Journal

Received August 30, 2021, accepted October 1, 2021, date of publication October 15, 2021, date of current version October 25, 2021.

Digital Object Identifier 10.1109/ACCESS.2021.3120629

Low Latency YOLOv3-Tiny Accelerator for Low-Cost FPGA Using General Matrix Multiplication Principle

TRIO ADIONO[1,2], (Member, IEEE), ADIWENA PUTRA[2], NANA SUTISNA[1,2], (Member, IEEE), INFALL SYAFALNI[1,2], (Member, IEEE), AND RAHMAT MULYAWAN[1,2], (Member, IEEE)

[1]Electrical Engineering Department, School of Electrical Engineering and Informatics, Institut Teknologi Bandung, Bandung, West Java 40116, Indonesia
[2]University Center of Excellence on Microelectronics, Institut Teknologi Bandung, Bandung, West Java 40132, Indonesia

Corresponding author: Infall Syafalni (infall@ieee.org)

This work was supported by the Research, Community Service, and Innovation (P2MI) School of Electrical Engineering and Informatics (SEEI), ITB Professor Research Program.

ABSTRACT This paper presents a comprehensive hardware accelerator architecture of YOLOv3-Tiny targeted for low-cost FPGA with a high frame rate, high accuracy, and low latency. The proposed accelerator implements all YOLO layers in hardware including zero pad layer, convolution layer, leaky ReLU layer, batch normalization layer, max-pooling layer, and up-sampling layer. The architecture is built based on data flow and control flow hybrid architecture. The data preparation and computation process work asynchronously using the data flow paradigm, while the overall governing process is controlled by proposed custom instruction set which adopts the principle of control flow paradigm. The principle of General Matrix Multiplication (GEMM) is adopted to compute the convolution process. We designed a GEMM processor using an optimum size of systolic array architecture. The systolic core is small and the overall architecture supports the multicore system, making it scalable to be implemented on larger size FPGAs. We also proposed a hardware architecture for mapping feature maps into matrix form for GEMM convolution which can save on-chip memory space. Lastly, we modified the original YOLO algorithm to further optimize it in our hardware. The modification includes reducing the bit precision to reduce memory space and bandwidth requirement, merging the normalization layer with the convolution layer to reduce arithmetic complexity, and adding a new DLQ layer to keep the bit size small while maintaining the accuracy. Based on the experimental results, our proposed design manages to achieve a frame rate of 8.3 FPS with the throughput of 31.5 GOPS, outperforming the same convolution computation that is performed by Ryzen 5 3600 CPU up to 69.3× in latency. Moreover, our proposed design also has the smallest clock cycle ratio up to 1.75× than other commercial accelerators. The system is useful and suitable for edge computing applications.

INDEX TERMS Accelerator, GEMM, low-cost FPGA, YOLOv3-Tiny.

I. INTRODUCTION

In recent years, the advancement of the Internet of Things (IoT) and the Artificial Intelligence (AI) technologies have opened a new type of problems. IoT has created a new paradigm of connectivity, where every device with an attached sensor can be connected to the cloud (internet). Meanwhile, artificial intelligent (AI) has started becoming a new emerging technology that can be applied in many fields. AI needs a lot of training data to build an accurate model. The data can be provided by the IoT applications through the sensors that can be used to improve services and products in industries. This combination does not only lead to the field of cloud computing, where AI is trained and deployed on the cloud, but it also raises more concerns as the following factors [1], [2]:

- **Latency**: AI is usually implemented in a real-time system. In the traditional cloud computing model, deploying AI on the cloud will introduce additional latency because the data needs to travel over the internet before it can be fetched into AI.
- **Bandwidth**: The increasing number of IoT will result in more data being generated. Transmitting all of the data will cause great burden on network bandwidth.

The associate editor coordinating the review of this manuscript and approving it for publication was Stavros Souravlas.

This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 License.
For more information, see https://creativecommons.org/licenses/by-nc-nd/4.0/

- **Availability**: AI cloud services rely on internet connectivity.
- **Security and Privacy**: Data generated by IoT might be considered as private data. Sending data over the internet makes it vulnerable to data stealing and loss of privacy.

In relation to the above mentioned concerns, people started to deploy AI directly on the edge device, where the IoT sensor is placed. This implementation, known as edge computing, also comes with potential trade-off problem between energy consumption and computing power. AI needs high computing power to deliver best results. On the other hand, IoT needs low energy consumption, which limits the computing power of the device. To overcome this problem, people started to develop edge friendly AI algorithm and also developing new specialized hardware architecture for AI (AI chip). In the current state of AI, people usually divide the AI process into training and inference processes [3]. Training the AI is usually computationally intensive and a lot of precision is needed, thus the training process is usually done on the cloud. On the other hand, the inference process, especially in a real-time application, needs to be done fast (with low latency), thus this process is usually done on the edge device.

One of the fields, where edge computing can be beneficial, is computer vision. It includes vehicle counting for traffic analysis, advanced driver-assistance system (ADAS), and face recognition system for security purposes. Several methods exist to implement computer vision solutions, but one of the most promising solutions came in the year 2012 when AlexNet won the ImageNet competition by achieving error of 15.3% [4]. AlexNet used a deep convolutional neural network which consists of convolutional layers, max-pooling layers, and fully connected layers. Since then, many solutions for computer vision problems built on a multi-layer structure consisting of convolutional layers, max-pooling layers, and fully connected layers.

In the following years, the solution for computer vision develops into a network known as Convolutional Neural Network (CNN). Some of the popular networks that had been developed are R-CNN [5], Fast R-CNN [6], and Faster R-CNN [7]. These R-CNN types of networks work in a similar approach. There are two main processes that working together. The first is the region proposal which will select the possible candidate to be categorized as an object in the picture. And, the second is a deep neural network that will be responsible for the object classification. The resource sharing between these two processes usually resulting in high computational load.

In order to lower the computational load, another approach known as YOLO (You Only Look Once) [8] had been developed. One of the major advantages of using YOLO is that it can predict the position of the object (bounding box) and the classification of objects in the same computation process and thus greatly reduce computational loads. The early version of YOLO is focused on low latency purposes with low accuracy trade-off. However, more recent version called YOLOv3 is targeted for both low latency and higher accuracy applications [9]. More specifically, there is also another variant called YOLOv3-Tiny, that is optimized for edge computing application.

Although computational load of YOLO is lower than its predecessor, the load is often still too high to be handled by a general-purpose CPU. Another candidate for hardware accelerator that can process high computational load would be GPUs. However, GPUs are known to be too costly in terms of energy consumption and their form factors are too large to be deployed on embedded system solutions [10], [11]. This paper addresses this problem by building YOLOv3-Tiny accelerator on a relatively low-cost (off the shelf) FPGA that can be easily deployed on edge computing applications.

One of the first works related to YOLOv3-Tiny accelerator was built on high-performance FPGA [12] which is still not suitable for low-cost application on edge computing solutions. Other works tried to build this accelerator using low-cost FPGA [13], but still suffer from the relatively high latency problems (532 ms). One of the most computationally expensive in YOLO is the 2D convolution process. Both of these works [12], [13] are using the same approach to compute the convolution process by using sliding windows. This approach is preferred because of its simplicity to be implemented and its practicality to be scaled in more complex design. However, the drawback of this approach is lack of flexibility when there are variations of kernel dimension and inefficiency. Moreover, the number of channels is growing large, since the computation is done per kernel per channel. The other approach for convolution is by using Generalized Matrix Multiplication (GEMM) where the kernel and feature map need to be converted to the matrix form [14]. By using GEMM, the variations of kernel dimension and size of channel are not problems because all the calculations are performed as matrix multiplications. The example of this approach is used by Google TPU [15] which uses the systolic core to compute the matrix multiplication. The biggest drawback of using GEMM is the need to convert kernel and feature map into the matrix form. The conversion process usually consists of complex memory access patterns and inefficient data storage because the data needs to be duplicated. The process of conversion is usually done in software by using a general purpose CPU which can significantly increase the latency, especially if the CPU is slow. Because of these reasons, the GEMM method is suboptimal for edge computing implementation.

The architecture of YOLOv3-Tiny accelerator is proposed in this paper built on GEMM principle using systolic core. However, unlike Google TPU that implements one big size of systolic core, our systolic core is smaller and it also supports multi-core architecture. It makes our architecture is easily scalable to other FPGAs. To address the problem of conversion into matrix form, our accelerator also provides mapping module that can map kernel and feature map into the matrix form. The result of this mapping is the compressed form of matrix which will save on-chip memory space. Also, this mapping process is pipelined into entire YOLO process and

as a result, it can reduce the latency of the accelerator. The main contributions of this work are as follows:

- We propose YOLOv3-Tiny hardware accelerator based on small size systolic core which is suitable for edge computing platform.
- We perform detail profiling of hardware dataflow based on input stationary dataflow. This dataflow can be used as a model to estimate the energy consumption for different workload.
- We propose hardware implementation for the compressed *im*2*col*() function which maps the input FMAP into the compress matrix form for GEMM computation. The compressed matrix form takes space $\approx$ 7.89 times less than direct/naive implementation of *im*2*col*().
- We perform detail profiling for each module in our accelerator, including the architecture and scheduling strategy to achieve fully pipelined architecture.

The rest of the paper is structured as follows. Section II presents the background knowledge and preliminary work that are required for implementing the YOLOv3-Tiny using hardware. In Section III, we describe our hardware dataflow which shows the data movement and memory hierarchy in our accelerator. In Section IV, we give the detailed explanation about our accelerator architecture. Section V explains about the processing flow between the host CPU and the accelerator, as well as the processing flow inside FPGA. Section VI discusses the implementation of our own custom instruction set. Section 42 discusses the implementation and the testing of our overall system. Section VIII contains several parts including: 1) an evaluation of our accelerator in the form of memory access analysis, 2) *im*2*col*() function analysis, 3) comparison with other AI accelerator in other platforms (CPU, GPU, & ASIC), and 4) comparison with other previous works. Finally, the paper is concluded in Section IX.

II. BACKGROUND

This section presents the background knowledge of our works. Subsection II-A discusses the overview of our YOLOv3-Tiny model and how we break the inference process into several batch layers. Subsection II-B discusses the convolution operation and the parameter naming which will be used throughout the paper. Subsection II-C and subsection II-D review the max pooling and upsampling operations in YOLO. Subsection II-E discusses the method to merge batch normalization computation into convolution process. Subsection II-F discusses the method that we used to reduce the bit precision and how it affected the performance of the system. Subsection II-G discusses the additional layer that we add into inference process as the consequence of reducing the precision. Subsection II-H discusses the method that we used to compute the convolution. Subsection II-I discusses the process of mapping FMAP (feature map) into the FMAP matrix (the term FMAP is equal to the activations in deep neural network [3]) using *im*2*col*() function.

A. YOLOv3-TINY MODEL

Our YOLO model performs prediction in 2 scales. First, by using 14 $\times$ 14 grid for big object detection, and second by using 28 $\times$ 28 grid for small object detection. The input image size will be 448 $\times$ 448 pixel with three RGB channels.

Table 1 shows the entire inference process of YOLOv3-Tiny. We will separate all this layers into several batch layers where each batch layer consist of only one convolution layer. As a result, we will have 13 batch layers. The accelerator will compute each batch layers sequentially, so only one batch layer can be computed at a time.

TABLE 1. YOLOv3-Tiny inference process.

Layer Batch	Layer	Input Channel	Output Channel	Filter Size	Input Fmap Size
1	Zero Pad Conv, Leaky-ReLU, BN Max Pooling (stride=2)	3	16	3x3	448
2	Zero Pad Conv, Leaky-ReLU, BN Max Pooling (stride=2)	16	32	3x3	224
3	Zero Pad Conv, Leaky-ReLU, BN Max Pooling (stride=2)	32	64	3x3	112
4	Zero Pad Conv, Leaky-ReLU, BN Max Pooling (stride=2)	64	128	3x3	56
5	Zero Pad Conv, Leaky-ReLU, BN Max Pooling (stride=2)	128	256	3x3	28
6	Zero Pad Conv, Leaky-ReLU, BN Max Pooling (stride=1)	256	512	3x3	14
7	Zero Pad Conv, Leaky-ReLU, BN	512	1024	3x3	14
8	Conv, Leaky-ReLU, BN	1024	256	1x1	14
9	Conv, Leaky-ReLU, BN	256	128	1x1	14
10	Up Sampling Concat Zero Pad Conv, Leaky-ReLU, BN	384	256	3x3	28
11	Conv, Leaky-ReLU, BN	256	18	1x1	28
12	Zero Pad Conv, Leaky-ReLU, BN	256	512	3x3	14
13	Conv, Leaky-ReLU, BN	512	18	1x1	14

We also do some modifications on our YOLO algorithm so it can fit well into the hardware. Some of our modifications include reducing the bit precision, merging batch normalization operation with convolutional layer, and add new quantization layer to preserve the bit width in computation.

B. CONVOLUTION OPERATION

Convolution (2D) is the main computing operation that will be accelerated by our accelerator. In this section we will introduce formally how the 2D convolution computation is executed and the parameters naming convention which we will used throughout this paper. Figure 1 shows the parameter naming which will be used in entire paper, it also illustrates the 2D convolution on multiple channel image (3D tensor) using multiple channel multiple filter (4D tensor). Mathematically, this convolution process can be described using the following equation:

$$
\begin{aligned}
O[k][p][q] = & \sum_{c=0}^{C-1} \sum_{r=0}^{R-1} \sum_{s=0}^{S-1} I[c][Up+r][Uq+s] \\
& \times F[k][c][r][s] + B[k] \\
& 0 \leq k < K, 0 \leq c < C, 0 \leq r < R, 0 \leq s < S \\
P = & \frac{H - R + 2pad}{U} + 1, Q = \frac{W - S + 2pad}{U} + 1,
\end{aligned} \tag{1}
$$

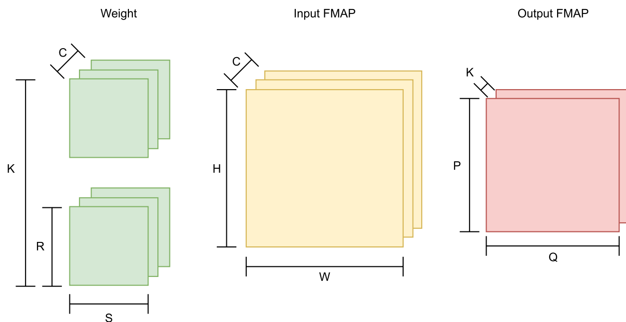


FIGURE 1. CNN computation and parameters naming.

where:

I = Input FMAP,
F = Filter weight,
B = Bias,
O = Output FMAP,
H = Input FMAP height,
W = Input FMAP width,
C = Input channel,
R = Filter kernel height,
S = Filter kernel width,
K = Output Channel,
P = Output FMAP height,
Q = Output FMAP width,
pad = Number of padding, and
U = Number of stride in convolution.

In our case, we use $pad = 1$ when R and S are 3, and use $pad = 0$ when R and S are 1. Thus, the input FMAP and output FMAP size will be the same ($H == P$ and $W == Q$). We also always use $U = 1$ for all batch convolution layer. For following sections, we will omitted the index notation of any tensor operation. For example, Equation (1) can be simply written as follows:

$$O = I \times F + B. \tag{2}$$

Notice that Equation (2) is referring to the same tensor multiplication in Equation (1).

C. MAX POOLING OPERATION

Max pooling is a down-sampling operation. It reduces FMAP size depending on the max pooling size and the stride value. Figure 2 shows the example of a max pooling operation that is performed on 4 × 4 FMAP. Max pooling takes the biggest value out of its window area and effectively reduces the FMAP height and width to the half. Mathematically, it can be represented as a $max()$ function from a set of number. Suppose that we have four values of FMAP in a set consisting $\{1, 5, 2, 6\}$ as Figure 2. Thus, the max pooling results:

$$max(\{1, 5, 2, 6\}) = \{6\}. \tag{3}$$

D. UpSampling OPERATION

Up-sampling operation is the reversal of max pooling operation. It increases FMAP size by doubling the height and width of FMAP. Figure 3 shows the example of up-sampling operation performed on 2 × 2 FMAP. This operation takes every pixel in FMAP and duplicates its area by 4 times. It is effectively doubles the height and the width of output FMAP. Mathematically, it can be represented as a $duplicate()$ function to a list of a set of number by 4 times as illustrated in Figure 3:

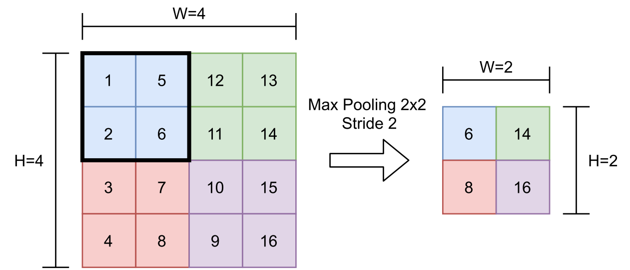


FIGURE 2. Max pooling operation illustration.

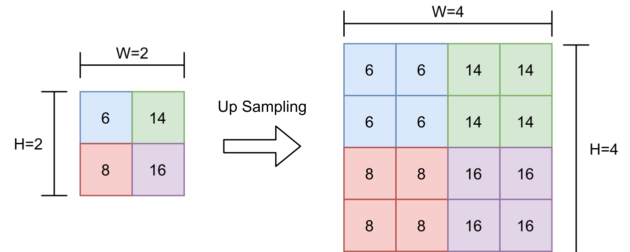


FIGURE 3. Up sampling operation illustration.

$$\begin{aligned} &duplicate(\{\{6\}, \{14\}, \{8\}, \{16\}\}) \\ &\quad = \{\{6, 6, 6, 6\}, \{14, 14, 14, 14\}, \\ &\qquad \{8, 8, 8, 8\}, \{16, 16, 16, 16\}\}. \end{aligned} \tag{4}$$

E. MERGING BATCH NORMALIZATION INTO CONVOLUTION

Batch normalization is the element wise operation that will be applied to the output of convolution layer in Equation (2). To reduce the number of operations, we will merge the batch normalization into the convolution process. This process can be achieved by rearranging the equations. Following is the equation of batch normalization that will be applied to each elements of matrix O:

$$O_{norm} = \frac{\gamma}{\sqrt{\sigma^2 + \epsilon}} O + \left(\beta + \frac{\gamma\mu}{\sqrt{\sigma^2 + \epsilon}}\right), \tag{5}$$

where:

O_{norm} = Output FMAP after normalized,
O = Output FMAP,
σ^2 = Estimated variance of O,
ϵ = Small constant to prevent numerical error,
μ = Estimated mean of O,
γ = Parameter to control variance of O_{norm}, and
β = Parameter to control mean of O_{norm}.

We can further simplify the equations by writing $A = \frac{\gamma}{\sqrt{\sigma^2+\epsilon}}$ and $D = \left(\beta + \frac{\gamma\mu}{\sqrt{\sigma^2+\epsilon}}\right)$, thus the Equation (5) becomes:

$$O_{norm} = AO + D. \tag{6}$$

Furthermore, we can substitute O from Equation (6) with expression from Equation (2) and the equations becomes:

$$O_{norm} = A\,(I \times F + B) + D. \tag{7}$$

By rearranging Equation (7), we get the convolution equation with new weight and new bias as the following:

$$O_{norm} = I \times F_{new} + B_{new}, \tag{8}$$

where $B_{new} = AB + D$ is a new bias and $F_{new} = AF$ is a new weight. By merging the batch normalization with convolution, we can reduce the resources as well as the computation time.

F. REDUCING PRECISION (QUANTIZATION)

Reducing the number of bits for either weight or feature map have several benefits. First, it can reduce the amount of data movement resulting in lower energy consumption. Reducing data movement can also increase throughput, since it reduces memory bandwidth requirements. Second, it also reduces the amount of on chip memory required for a given number of weights, activations, and/or partial sums [3]. In our version of YOLOv3-Tiny, we use reference weight with 32-bit floating-point precision.

We use linear symmetrical quantization method to convert the parameter from 32-bit floating point into 8-bit signed integer. We choose linear quantization to reduce energy consumption and memory space during the computation time. By choosing linear quantization, we manage to do the multiplication in 16-bit and accumulation in 32-bit. By choosing symmetrical method, we eliminate the need to use zero point during quantization, thus it reduces the number of operation. The first step in quantization is to find the scale then use the scale to find the quantized value. The overall process of quantization can be explained as follows [20]:

$$scale = \frac{max - min}{2^b - 1}$$
$$q_x = round\left(\frac{x}{scale} + zp\right) \tag{9}$$

where:

x = Parameter to be quantized (floating),
max = Maximum value of parameter in 1 tensor (floating),
min = Minimum value of parameter in 1 tensor (floating),
b = Number of bits (we used 8-bit),
zp = Representation of zero floating in quantized value,
$scale$ = Scaling parameter in 1 tensor (floating), and
q_x = Quantized value of parameter (8-bit integer).

In Equation (9), we use $round()$ function to round the value to the nearest integer. In our case, the zp value will be zero, because zero floating point will be represented as zero in 8-bit signed integer (symmetrical quantization). Our version of YOLO reduces the precision of weight and FMAP up to 8 bits, while maintaining the mAP of 75% tested on our custom dataset (derived from VOC 2007 dataset). In real VOC 2007 dataset, the total number of image is 9963 images. We took 2665 images from the real VOC 2007 dataset, used the 2400 images for training and the remaining 265 images for testing. We measured that we only got 2% drop in mAP (in original YOLOv3-Tiny model, we got 77% mAP when tested against our custom dataset).

G. DLQ LAYER

The inference process in one batch layer can be seen in Figure 4. The process consists of four layers such as convolution layer, batch normalization layer, activation function layer, and max pooling. The output of max pooling usually will be used by similar inference process in the next batch layer.

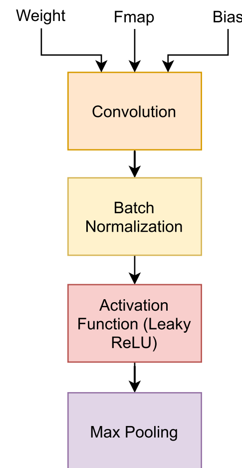


FIGURE 4. Typical inference process in one batch layer.

Because we apply quantization (for FMAP, weight, and bias), the inference process in one batch layer needs to be modified in order to preserve the bit precision. The modified inference process can be seen in Figure 5. In order to prevent more loss in accuracy, the activation function operation will be done in 32-bit floating point number. Because of this, two additional layers are required: (1) the dequantization layer before the activation function layer and (2) the quantization layer after the activation function layer. The dequantization layer will dequantize back the result of batch normalization into 32-bit floating point, while the quantization layer will quantize the result of activation function layer back to 8-bit signed integer.

Similar to Section II-E where we merge the batch normalization layer and convolution layer, the activation function layer can be merge with both dequantization layer and quantization layer by rearranging the equations. The quantized value of a number, as we can see from Equation (9), is obtained by dividing the number with its

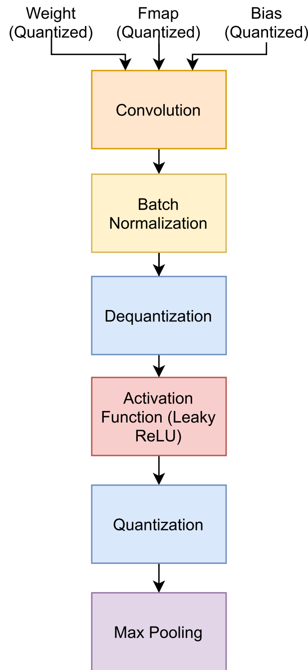


FIGURE 5. Modified inference process in one batch layer.

scale. To dequantize the value of a number, we just need to reverse the process by multiplying the number with its scale.

$$dq_x(x) = x \times scale \tag{10}$$

Mathematically, Leaky ReLU formula can be written as following

$$LeakyReLU(x) = \begin{cases} x & \text{if } x \geq 0 \\ ax & \text{otherwise} \end{cases} \tag{11}$$

where a is scaling coefficient with value $0 < a < 1$. The output of Equation (11) will be quantized using the Equation (9). Combining the Equations (9), (10), and (11), we obtain the DLQ (Dequantization - Leaky ReLU - Quantization) formula as following:

$$DLQ(x) = \begin{cases} x\ scale_{+} & \text{if } x \geq 0 \\ x\ scale_{-} & \text{otherwise.} \end{cases} \tag{12}$$

By merging these three equations, we can reduce both resource used and computation time.

H. CONVOLUTION AS GENERAL MATRIX MULTIPLICATION

Convolution is often done by using sliding window approach, where there are separate hardware block for each kernel. In modern CNN architecture such as YOLO, the kernel size and dimension often vary across multiple batch layers. Designing separate hardware block for each kernel increases resource utilization and makes the hardware become rigid. Transforming convolution as generalized matrix-matrix multiplication (GEMM) makes the hardware design kernel size agnostic [17]. To use GEMM in convolution, both the weight and FMAP need to be transformed into matrix form (2D tensor). The process of transforming weight tensor (4D tensor) into matrix is quite straight forward, we just need to reshape the weight dimension from $F[K][C][R][S]$ to $F[K][CRS]$. For FMAP, the transformation process into matrix form will be more complex. It will expand the size of FMAP and duplicate some data from the initial FMAP 3D tensor. The size of FMAP matrix also depends on dimension of F, the *stride* value, and *pad* value. Formally, the name of function to transform FMAP into matrix FMAP is known as *im2col*() function. The details of this function will be explained in next section.

Figure 6 illustrates the convolution process using GEMM. In the example, the weight with initial shape $F[K = 1][C = 1][R = 2][S = 2]$ is reshaped to $F[K = 1][CRS = 4]$. While the FMAP with initial shape $I[C = 1][H = 3][W = 3]$ is transformed to FMAP matrix with shape $I[CRS = 4][PQ = 4]$. Once the weight and FMAP in matrix form, we can compute 2D convolution as matrix multiplication. The resulting matrix multiplication has dimension of $O[K = 1][PQ = 4]$.

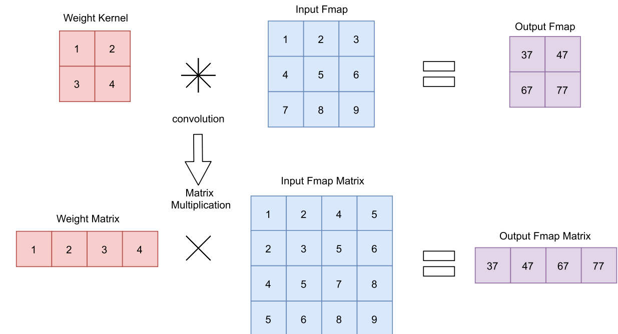


FIGURE 6. 2D convolution using matrix multiplication.

I. Im2col FUNCTION

Im2col (*im2col*()) function performs the transformation of 3D tensor $I[C][H][W]$ to 2D tensor matrix. The matrix will have the dimension of $I[CRS][PQ]$, where RS is the size of filter kernel and PQ is the size of output FMAP. The size of the output FMAP is derived from HW size of input FMAP, which also affected by the filter size, stride and padding value (Refer to Equation (1) for complete relationship between HW and PQ).

More generally, *im2col*() function is described in Algorithm 1. As shown, there will be an increase in total weight size of matrix compare to its 3D tensor form. The ratio between matrix and 3D tensor FMAP size is $\frac{RSPQ}{HW}$. In the example from Figure 6, the input FMAP matrix size increase with the ratio of $\frac{RSPQ}{HW} = \frac{2\times2\times2\times2}{3\times3} = \frac{16}{9}$. In our case, we select the stride and padding value, thus we have $HW == PQ$. Because of this, the output matrix size is RS times larger than the original 3D tensor FMAP.

Transforming the weight directly using *im2col*() function will increase the FMAP matrix size by RS times larger (in

Algorithm 1 Im2col Function

```
1: /* Variables: */
2: fmap[C][H][W]
3: mat_fmap[CRS][PQ]
4:
5: for c=0 in range(0,C) do
6:     for p=0 in range(0,P) do
7:         for q=0 in range(0,Q) do
8:             for r=0 in range(0,R) do
9:                 for s=0 in range(0,S) do
10:                    mat_fmap[c*R*S+r*S+s][p*Q+q] = fmap[c][U*p+r][U*q+s]
11:                end for
12:            end for
13:        end for
14:    end for
15: end for
```

most cases $RS = 9$). This will cause a problem, especially in edge computing application, because of limited on-chip memory capacity. If on-chip memory capacity is not enough to store all FMAP matrix, then after applying $im2col()$ function, we need to store the FMAP matrix in off-chip memory. It will increase total latency and energy consumption. To address this problem, we will use tiling strategy to transform a tile of FMAP into a tile of matrix FMAP and store the tile in compressed matrix form. Thus, we can save on-chip memory space. By using this strategy, we managed to transform FMAP into FMAP matrix without storing the matrix FMAP in off-chip memory. The detail of this strategy will be explained in Section IV.

III. PROPOSED HARDWARE DATAFLOW

Dataflow is one the most important aspect in designing AI accelerator. Dataflow defines some specific rules for controlling activity of an accelerator including the ordering operation, scheduling process in temporal and spatial space, tiling strategy, and the data orchestration across multiple memory hierarchy in datapaths. The Dataflow usually is described using the loop nest pseudocode.

Figure 7 shows the abstraction of memory level in our accelerator. The DDR memory is located outside the FPGA chip, L1 memory located inside the FPGA chip in the form of BRAM, while L0 memory also located inside the FPGA chip in the form of registers near the processing element. In the term of latency and energy cost, L0 memory is the fastest, L1 memory is medium, and DDR memory is the slowest. However, in the term of memory capacity, L0 memory is the smallest, L1 memory is medium, and DDR memory is the largest. Our goal is to schedule the data movement in each memory level, thus we can minimize the latency and energy consumption of our accelerator.

Because the sizes of L1 memory and L0 memory are smaller than DDR memory, we need to tile the FMAP from DDR memory to L1 and L0 memories. Figure 8 shows the tiling strategy for the case of the first batch layer. There are 4 parameters, which will dictate our tiling and dataflow process: H1, W1, NH1, NW1. Each values of these parameters is derived from following equations:

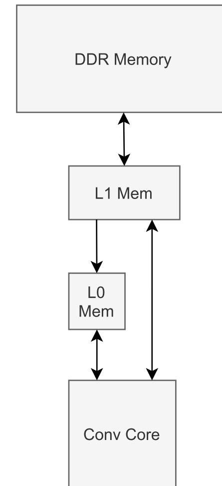


FIGURE 7. Memory hierarchy in accelerator.

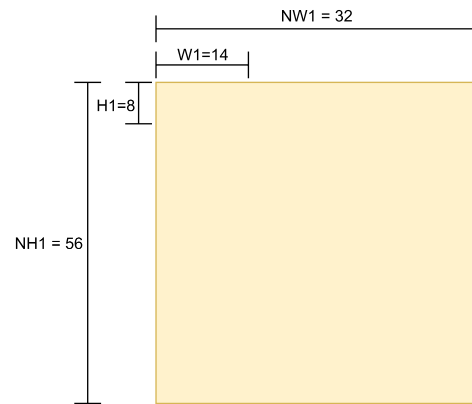


FIGURE 8. Tiling strategy for first batch layer.

$$\begin{aligned} W1 &= 14 \\ NW1 &= \frac{W}{W1} \\ H1 &= \frac{MemSize}{W \times C} \\ NH1 &= \frac{H}{H1} \end{aligned} \tag{13}$$

where:

$W1$	= Width for systolic array,
$NW1$	= Number of W1 in FMAP width,
$H1$	= Number of FMAP rows store in L1 memory,
$NH1$	= Number of H1 in FMAP height,
H	= Input FMAP height,
W	= Input FMAP width,
C	= Input channel, and
$MemSize$	= Allocated L1 memory for FMAP.

We choose the tiling size of $W1$ to be 14. The reason for this is because the smallest FMAP width that we need to process is 14 pixel. If we do the computation in smaller size input FMAP, choosing larger value for more than 14 will be redundant. The other parameters ($NW1, H1, NH1$) will be vary across different batch layer. For the case of first batch layer, input FMAP dimension is $I[C = 3][H = 448][W = 448]$. Applying these properties to Equation (13), we will get the parameter value in Figure 8.

We use input stationary dataflow to optimize the reuse of input data (each pixel in IFMAP only need to be read once in every batch layers). L1 memory acts as global buffer for all data types (IFMAP, weights, bias, and partial sum) and L0 memory acts as small memory to store the IFMAP during computation. Algorithm 2 shows the dataflow for our accelerator. There are 8 loop level on our dataflow. The outermost loop in line 10 describes the loop iteration over $NH1$ parameter. During this loop, we do the read operation for input FMAP (described in line 11) in which the accelerator reads the first $H1$ rows of FMAP in every channel and stores it in L1 memory. The second loop in line 12 describes the loop iteration over C parameter. During this loop, the accelerator reads the KRS value of weight in the first channel from DDR memory to L1 memory (described in line 13). The next two loops (line 14 and line 15) iterate over $H1$ and $NW1$ parameters. During these loops, we will fill L0 memory with FMAP value from L1 memory. This process is described in line 16 using the *Fill*() function. In addition, the *Fill*() function also implements padding and *im2col*() function transforms the input FMAP into the matrix form. The next three loops (line 17, line 19, and line 20) are "**for parallel**" loops. It means that, we will implement parallel architecture for these loops in hardware. Loops in line 19 and 20 describe the size of our systolic array, which is $RS \times W1 == 9 \times 14$. The loop in line 17 describes the number of systolic core that will be implemented in the hardware (where each core consist of 9×14 systolic array). The loop in line 21 describes the MAC operations over $K2 - K1$ iterations. The value of $K2$ and $K1$ are the bounds in which each core does the computation (*e.g.*, for the case of $Ncore = 2$ and $K = 16$, the first core will compute from $K1 = 0$ until $K2 = 7$ and the second core will compute from $K1 = 8$ until $K2 = 15$). Finally, the final partial sum value (output FMAP) will be written out to DDR memory in the end of first loop (line 31).

Algorithm 2 Input Stationary Dataflow Loop Nest

```
1:  /* Memory Level: */
2:  ifmap[C][HW] // DDR Memory
3:  ofmap[K][HW] // DDR Memory
4:  weight[C][K][RS] // DDR Memory
5:  ifmap_buffer[C][H1*W] // L1 Memory
6:  weight_buffer[K][RS] // L1 Memory
7:  ofmap_buffer[K][H1*W] // L1 Memory
8:  ifmap_pe[RS][W1] // L0 Memory
9:
10: for nh1 in range(0, NH1) do
11:   ifmap_buffer = LoadFmap(ifmap, C, H1, W, nh1)
12:   for c in range(0, C) do
13:     weight_buffer = LoadWeight(weight, K, RS, c)
14:     for h1 in range(0, H1) do
15:       for nw1 in range(0, NW1) do
16:         ifmap_pe = Fill(ifmap_buffer, RS, W1, nw1,
            h1, c)
17:         for parallel core in range(0, Ncore) do
18:           K1,K2 = AssignBound(K, core)
19:           for parallel rs in range(0, RS) do
20:             for parallel w1 in range(0, W1) do
21:               for k in range(K1, K2) do
22:                 idx = AddrIdx(w1,nw1,h1,nh1)
23:                 ofmap_buffer[k][idx] +=
                    weight_buffer[k][rs] *
                    ifmap_pe[rs][w1]
24:               end for
25:             end for
26:           end for
27:         end for
28:       end for
29:     end for
30:   end for
31:   ofmap = WriteFmap(ofmap_buffer, K, W1, nh1)
32: end for
```

IV. PROPOSED HARDWARE ARCHITECTURE

This section explains the detail of proposed hardware accelerator architecture. First, we give an overview of our entire system in Subsection IV-A. From that section, our architecture can be separated into five sections. The first section is Input Bias Path, which will be explained in greater detail in Subsection IV-B. The second section is Input FMAP Path, which will be explained in Subsection IV-C. Next, the Input Weight Path will be explained in Subsection IV-D. The fourth section is the Matmult Core, where we will compute the process of convolution in matrix multiplication. This will be explained in Subsection IV-E. The last section will discuss the post processing after the convolution that will be explained in Subsection IV-F.

A. SYSTEM OVERVIEW

Figure 10 shows the top level view of the entire system. From the top level, the host CPU will give command to accelerator through AXI Lite protocol communication. Essentially, there are only two parameters that host CPU needs to provide for the accelerator. The first is the DDR's address location of instruction and the second is the total number of instructions to be read from DDR. The total number of instructions corresponds to the total number of batch layers. As explained in Section 1, for our model, there will be 13 instructions that correspond to 13 batch layers. After the host CPU gives a start signal the accelerator, all instructions will be loaded to the BRAM instructions and the process of computation for each

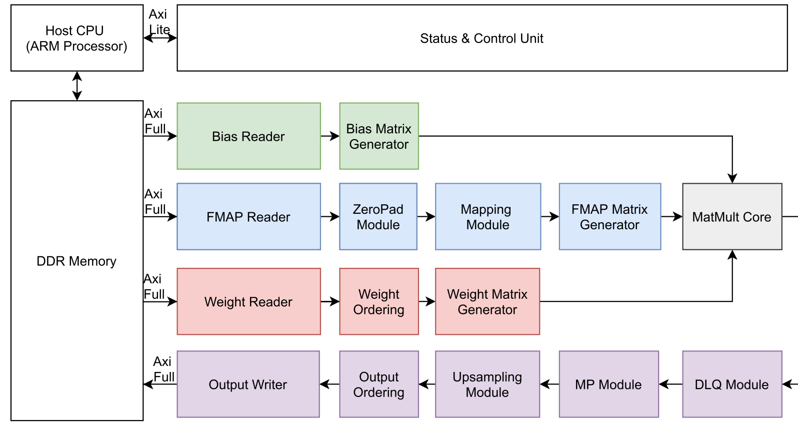


FIGURE 9. Architecture of the accelerator.

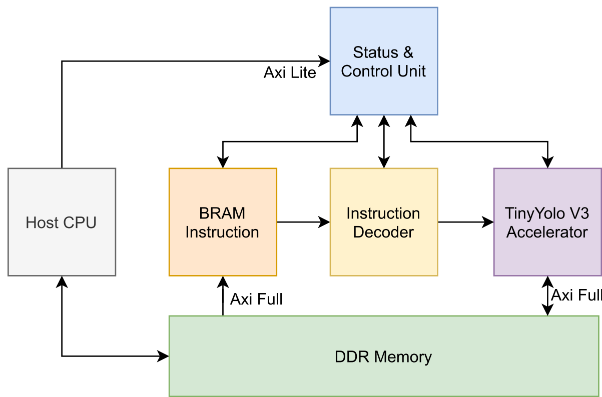


FIGURE 10. Top level of the accelerator.

batch layer will be started. For each batch layer, the process starts from reading the instruction from BRAM instructions, decoding the instruction, and fetching it to the YOLOv3-Tiny Accelerator.

Figure 9 provides the inside of YOLOv3-Tiny Accelerator. The accelerator consists of several modules that can be categorized into five sections:

- **Input Bias Path**, consists of bias reader module and bias matrix generator module.
- **Input FMAP Path**, consists of FMAP reader module, zero pad module, mapping module, and FMAP matrix generator module.
- **Input Weight Path**, consists of weight reader module, weight ordering module, and weight matrix generator module.
- **Matmult Core**, consists of 9×14 systolic array to compute the matrix multiplication and matrix addition.
- **Output FMAP Path**, consists of DLQ module, max pooling (MP) module, up-sampling module, output ordering module, and output writer module.

To achieve fully pipeline architecture, each module in the accelerator is separated by FIFO.

For I/O DDR memory access, we use AXI Full protocol with the accelerator as the master. By using this, we eliminated the need of host CPU to manage the data transfer process between DDR and accelerator. Thus, our host CPU is completely free of any task. There are three input paths to our accelerator and one output path from our accelerator. The three input paths are each for bias, FMAP, and weight, while the output path is used to write the result to DDR.

B. INPUT BIAS PATH

Input bias path consists of bias reader module and bias matrix generator module. The bias reader module's task is to produce the DDR address from which the bias data will be read. It also handle all signals related to AXI Full protocol. The bias matrix generator module's task is to form the bias matrix that will be used by Matmult Core. The bias matrix dimension will be $B[K][14]$, where K is number of output channel and 14 is corresponding to the width ($W1$) of our systolic array. The detailed explanation including the architecture and the timing of bias reader module and bias matrix generator module will be explained in following sections.

1) BIAS READER MODULE

The bias reader module consists of the address generator FSM to get bias from the DDR memory. The address generator will generate the DDR address and pass the address to AXI Full master (as shown in Figure 11). AXI Full master will handle the process of AXI Full protocol and get the data from DDR. The bias data will be read during read operation and it will be written to the next FIFO during write operation. During one

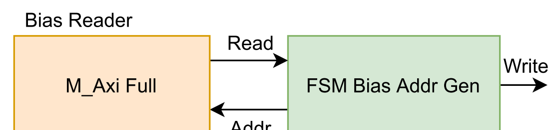


FIGURE 11. Architecture of bias reader.

read operation, we get the total of 16 numbers of bias. This number is chosen because it is the greatest common divisor of bias number across all batch layers. It is also required for AXI Full protocol [18], where the bit width of data transfer must be 2^n (where n minimum is 3 and maximum is 10). In our case, each bias is 32 bits, thus the total of 512 bits (16 words) of data are transferred at a time. The process of reading bias happens once outside the loop nest of Algorithm 2. This module manages to achieve execution process with 1 clock cycles of initiation interval and the latency of 3 clock cycles. The detail of scheduling process can be seen on Figure 12.

Bias Reader (II=1, Latency=3)								
Data In (16 words width)	Clock Cycles							
	C0	C1	C2	C3	C4	C5	C6	C7
1st		Addr	Read	Write				
2nd			Addr	Read	Write			
3rd				Addr	Read	Write		
4th					Addr	Read	Write	
5th						Addr	Read	Write

FIGURE 12. Operation scheduling in bias reader module.

2) BIAS MATRIX GENERATOR MODULE

This module generates the bias matrix with the size of $B[K][14]$, where K is the number of output channel. Figure 13 shows the architecture of bias matrix generator module. The read operation happens once every 16 clock cycles, where in every clock cycles the bias data will be duplicated 14 times before they are written out to the FIFO. Figure 14 shows the example of $B[16][14]$ bias matrix that will be produced when K is equal to 16.

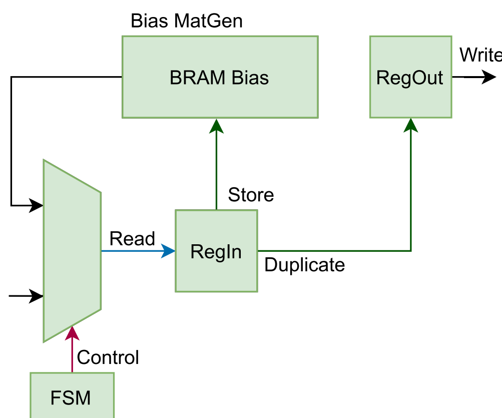


FIGURE 13. Architecture of bias matrix generator.

The bias matrix generator module manages to execute with 1 clock cycles initiation interval and the total latency is 4 clock cycles. The detail of scheduling process for this module can be seen on Figure 15. The process starts from control operation, which acts as the control unit. It determines which operation will be executed in every clock cycle. Following that, the read operation will execute to read 16 words of bias from input FIFO and will store the read results in RegIn registers. As a note, the read operation can be performed from input FIFO or BRAM bias. Store and duplicate operation can be executed on the same time. The store operation stores the 16 words of bias in BRAM, while the duplicate operation duplicates the fist word of bias (out of 16) 14 times in single clock cycles before it will be written out to the output FIFO during the write operation. In the next clock cycles, the duplicate operation will be performed again up to 16 times before this module do the next read operation.

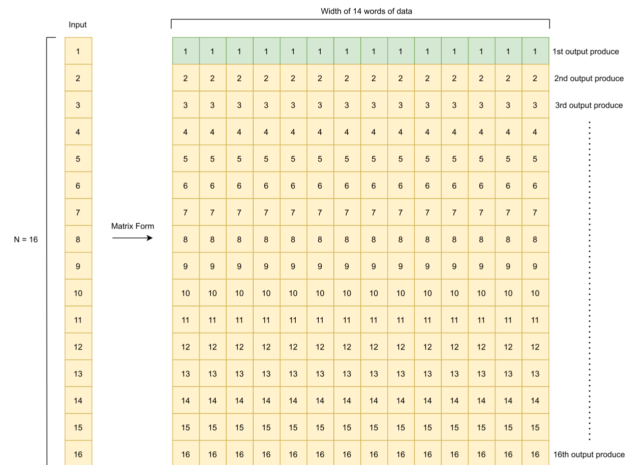


FIGURE 14. The process of bias matrix formation.

Bias Matrix Generator (II=1, Latency 4)									
Data In (16 words width)	Clock Cycles								Data Out (14 words width)
	C0	C1	C2	C3	C4	C5	C6	C7	
1st		Control	Read	Store/ Duplicate	Write				1st
			Control		Duplicate	Write			2nd
				Control		Duplicate	Write		3rd
					Control		Duplicate	Write	4th
						Control		Duplicate	5th
							Control		6th
								Control	7th

FIGURE 15. Operation scheduling in bias matrix generator module.

C. INPUT FMAP PATH

Input FMAP path consists of FMAP reader module, zero pad module, mapping module, and FMAP matrix generator module. In the term of Algorithm 2, FMAP reader module performs the *LoadFmap*() function while zero pad module, mapping module, and FMAP matrix generator module perform the *Fill*() function. The FMAP reader module's task is similar to bias reader module. It produces DDR address and handles all related signals for AXI Full protocol. Zero pad module task adds zero padding surrounding the image in every channel. Not every layer needs zero padding. In some cases, the zero padding process can be skipped. Mapping module and FMAP matrix generator module perform the *Im2col*() transformation of FMAP (as explained in Section II-I). Mapping module will produce the compress matrix form of FMAP tile, while FMAP matrix generator

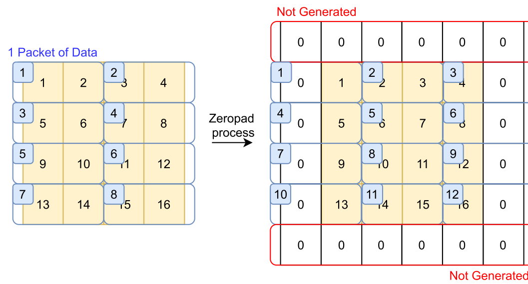


FIGURE 16. Illustration of zero padding process.

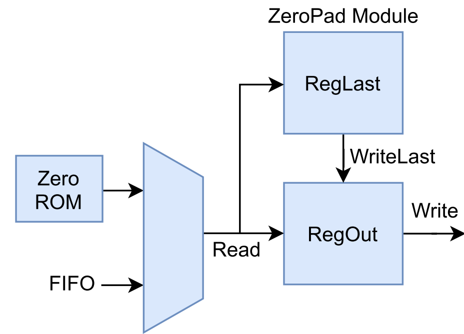


FIGURE 17. Architecture of ZeroPad module.

will expand the compress matrix into the large FMAP matrix where the convolution will be performed. For our accelerator, both mapping module and FMAP matrix generator are specialized to perform $Im2col()$ function with $U = 1$ and $RS = 3 \times 3 = 9$.

1) FMAP READER MODULE

FMAP reader module has similar architecture and functionality to the bias reader module in Figure 11. The scheduling process for this module is also similar to Figure 12. For one read operation, we get 16 adjacent value of FMAP in one channel. We will refer this 16 adjacent values as one data packet. The number 16 is chosen based on the requirement of AXI Full protocol [18]. The reading pattern of FMAP follows the loop nest in Algorithm 2. If we describe the input FMAP as $I[c][h][w]$, the reading order starts from index $w = 0$ until $w = W - 1$, index $h = 0$ until $h = H1 - 1$, and index $c = 0$ until $c = C - 1$, respectively. We will repeat this process $NH1$ times. For example, in the next reading process, we will start from index $h = H1$ until $h = 2H1 - 1$.

2) ZeroPad MODULE

This module will handle the process of padding the FMAP with zero value. To simplify the explanation, we use smaller size of FMAP as an example. Figure 16 illustrates this process with example of FMAP size 4×4 (where in our implemented design, the input FMAP size on the first batch layer is 448×448). Assume that the stream wide is two words, thus every packet of data received by this module is always two values at a time (while in our real implementation, the data packet has 16 words of values). In the Figure 16, the packets are shown with time-stamp number. This time-stamp shows the order of packet that will be received (left side of Figure 16) and produced (right side of Figure 16) by this module. The output is the zero padding result of FMAP except the top and bottom padding of the FMAP. The top and the bottom paddings are not generated because they will be generated later in the FMAP matrix generator module. By this way, we can reduce the bandwidth of data transfer between modules and also save on-chip memory spaces.

Figure 17 shows the internal architecture of ZeroPad module. It consists of mux in the beginning to choose the target of read operation. It is either to read from the input FIFO or the Zero ROM which stores the zero values. The read operation will split the packet from FIFO, where one word from data packet is stored in RegLast registers and the other one is stored in RegOut registers. The RegLast value is initialized by zero every time there is changes in FMAP row. The WriteLast operation is happening on the same time with the Read operation. The WriteLast operation will merge the values from the RegLast registers into the RegOut registers. Then, the Write operation will write out the RegOut register value to the output FIFO. Figure 18 shows the detail of operation scheduling in ZeroPad module. The module manages to achieve 1 clock cycles initiation interval with the latency of 2 clock cycles.

ZeroPad Module (II=1, Latency = 2)							
Data In (16 words width)	Clock Cycles						
	C0	C1	C2	C3	C4	C5	C6
1st		Read/ WriteLast	Write				
2nd			Read/ WriteLast	Write			
3rd				Read/ WriteLast	Write		
4th					Read/ WriteLast	Write	
5th						Read/ WriteLast	Write

FIGURE 18. Operation scheduling in ZeroPad module.

3) MAPPING MODULE

This module purpose is to transform the data packet from previous module into the suitable data form for FMAP Matrix Generator module. We called this suitable data format as compressed matrix form. The size of compressed matrix form depends on systolic array width size (which also has the tile size $W1$) and the kernel size (RS). Continuing the output of zero padding process from Figure 16, we use $RS = 9$ with systolic array width size of 2 ($W1 = 2$). Figure 19 shows the example illustration of mapping process from the previous module. The figure describes the input and the output results of this module along with the time-stamp to show the order of data received and produced. Notice that the input stream width is 2 words and the output stream width is 4 words. The output stream width was chosen to adjust with the systolic array width size. For the case of $U = 1$ and $RS = 9$, we choose the output stream width of $W1 + 2$.

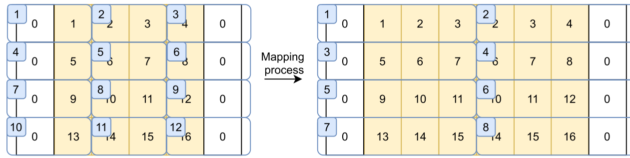


FIGURE 19. Illustration of mapping process.

For this example, because $W1 = 2$, the output stream width is 4 words. In our real implementation, we choose $W1 = 14$, thus output stream width is 16 words.

The procedures of mapping input to output in this module are as follows:

1) Concatenate the input data packet with stored data, thus it matches the output stream size.
2) Duplicate and store the overlapped areas between output data packet to handle the halos (cross tile dependencies between computing tile in convolution).

Figure 20 shows the internal architecture of Mapping module. It consists of three registers that interchange the data to concatenate and store the overlapped area between output data packets. Figure 21 shows the detail of operation scheduling in Mapping module. It manages to achieve initiation interval from 1 clock cycle to 3 clock cycles. There are total 5 operations that will be performed in this module. The Read operation reads the data packet from input FIFO into RegIn register. Store and StoreIn operations duplicate the data from RegIn registers into RegTemp and RegOut registers. StoreTemp operation will merge the data from RegTemp registers into RegOut registers. And, Write operation writes the data packet from RegOut registers into the output FIFO.

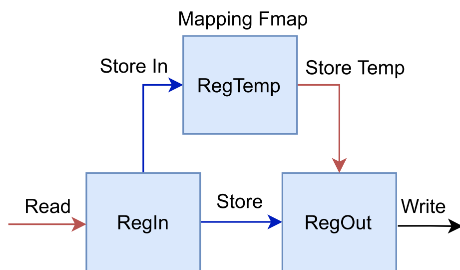


FIGURE 20. Architecture of mapping module.

4) FMAP MATRIX GENERATOR MODULE

FMAP matrix generator module will expand the compress matrix form to large FMAP matrix. Continuing example from previous module, Figure 22 shows the example how the output from mapping module in Figure 19 will be transformed into fully FMAP matrix which will be computed by the systolic array. The time-stamp on input and output data show the order of data being received or produced. The words size of each output data packet shows the width of systolic array, while the height of output matrix shows the height of systolic array. For the example in Figure 22, the systolic array size is 9 by 2 (while in our real implementation, the systolic array size is 9 by 14).

MappingFmap Module (II=1, Latency 3)								
Data In (16 words width)	Clock Cycles							
	C0	C1	C2	C3	C4	C5	C6	C7
1st		Read	Store/ StoreIn	Write				
2nd			Read	Store/ StoreIn/ StoreTemp	Write			
3rd				Read	Store/ StoreIn/ StoreTemp	Write		
4th					Read	Store/ StoreIn/ StoreTemp	Write	
5th						Read	Store/ StoreIn/ StoreTemp	Write

FIGURE 21. Operation scheduling in mapping module.

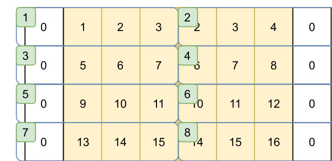


FIGURE 22. Input data and output produce from FMAP matrix generator module.

The internal architecture for this module is shown in Figure 23. There are total of five operations in this module. Store operation reads the data from input FIFO and stores it into the BRAM FMAP. Read operation reads data either from

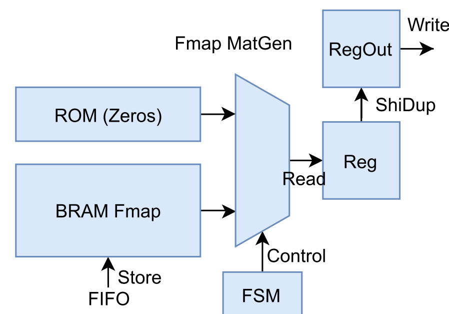


FIGURE 23. Architecture of FMAP matrix generator.

Fmap Matrix Generator (II=1, Latency 5)																		
Data In (16 words width)	Clock Cycles																	Data Out (14 words width)
	C0	C1	C2	C3	C4	C5	C6	C7	C8	C9	C10	C11	C12	C13	C14	C15	C16	
1st		Control	Store	Read	ShiDup	Write												1st
2nd			Control	Store		ShiDup	Write											2nd
3rd				Control	Store		ShiDup	Write										3rd
4th					Control	Store	Read	ShiDup	Write									4th
						Control			ShiDup	Write								5th
							Control			ShiDup	Write							6th
								Control		Read	ShiDup	Write						7th
									Control			ShiDup	Write					8th
										Control			ShiDup	Write				9th
5th											Control		Read	ShiDup	Write			10th
												Control	Store		ShiDup	Write		11th
													Control			ShiDup	Write	12th

FIGURE 24. Operation scheduling in FMAP matrix generator module.

ROM (filled with zero values) to produce the top and bottom paddings, or from BRAM FMAP. ShiDup operation is shift and duplicate operations to form the FMAP matrix as we have explained in Section II-I. Write operation writes the result into output FIFO and Control operation schedules the entire operation to determine whether the read operation reads the data from the ROM or the BRAM FMAP.

The detailed process of FMAP matrix formation in this module can be seen on Figure 25, using the example from Figure 22. The process is split into four parts. In the first process (Figure 25(a)), the first four data from input FIFO have been stored in BRAM FMAP (it tooks the total of 4 clock cycles). While in the same process of storing data to BRAM, this module starts write operation to produce the output with the time-stamp 1 to 3, which only need the zero value from the ROM (it took the total of 3 clock cycles). To produce the output with time-stamp 4 to 6, the module will perform Read operation from BRAM FMAP (specifically to read the input time-stamp 1 from BRAM) and perform ShiDup and Write operations. The similar process also happens to produce the output time-stamp 7 to 18. Figure 24 shows how the operation to produce the first 12 output from Figure 25 will be scheduled. This module manages to achieve scheduling with 1 initiation interval and the total latency is 5 clock cycles.

In the second process (Figure 25(b)), to produce the output with the time-stamp 19 to 21, the module performs Read operation from BRAM FMAP to get the input time-stamp 1. After performing ShiDup and Write operations to produce output time-stamp 19 to 21, the input time-stamp 1 in BRAM won't be used anymore and the module will perform Store operation to overwrite the input time-stamp 1 in BRAM with new input time-stamp 5. By this way, the BRAM spaces can be minimized by storing the needed data only. The process to produce the output with the time-stamp 22 to 36 also follows in similar way. The input time-stamp 2 in BRAM will also be overwrite by input time stamp 6 once the output time-stamp 28 to 30 have been produced. The third process (Figure 25(c)) also follows the same procedure to produce output with the time-stamp 37 to 54. The input time-stamp 3 and 4 in BRAM will also be overwritten by the input time-stamp 7 and 8. In last process (Figure 25(d)), no further data in BRAM will be overwritten. It only produces the output based on the input time-stamp 5 to 8 and the zero values from the ROM.

D. INPUT WEIGHT PATH

Input weight path consists of weight reader module, weight ordering module, and weight matrix generator module. In the term of Algorithm 2, these 3 modules perform the $LoadWeight()$ function. Weight reader module task is similar to bias reader module and FMAP reader module. Weight ordering module will convert the data format, thus it will be easy for the weight matrix generator module to form the weight matrix. Weight matrix generator module will produce the weight matrix $F[K][RS = 9]$, where K is the number of output channel and $RS = 9$ corresponds to the row size of systolic array. The detailed explanation for each modules will be explained in following sections.

1) WEIGHT READER MODULE

Weight reader module has similar architecture and functionality to Bias Reader Module in Figure 11. The operation scheduling is also similar to Figure 12. It accesses weight form DDR memory through AXI Full protocol [18]. It has input and output streams widths of 16 words, thus in one access, it can get 16 adjacent weights values from DDR. As Equation (1), the default way to store weight is by the following index order of tensor $F[k][c][r][s]$. In order to fit with our proposed dataflow, we will rearrange the storing order of weight in the DDR to $F[c][k][r][s]$.Since our L1 memory will only store weight per $F[c][r][s]$, this module will repeatedly read weight from the DDR as many as $C \times NH1$ times by the dataflow in Algorithm 2. Although this means we access the same weight $NH1$ times, it only affects a little in the overall bottleneck. The reason is because the weight needs to wait for the FMAP data before the convolution process is started. As Figure 9, the pipeline stage of input FMAP path is 4 stages, while the input weight path has only 3 pipeline stages. This means at the end of pipeline stage, the weight needs to wait for the FMAP data before the MatMult Core starts the computation. While waiting, the Weight Reader module can start reading the data from the beginning again. In addition, this access pattern reduces the overall on-chip

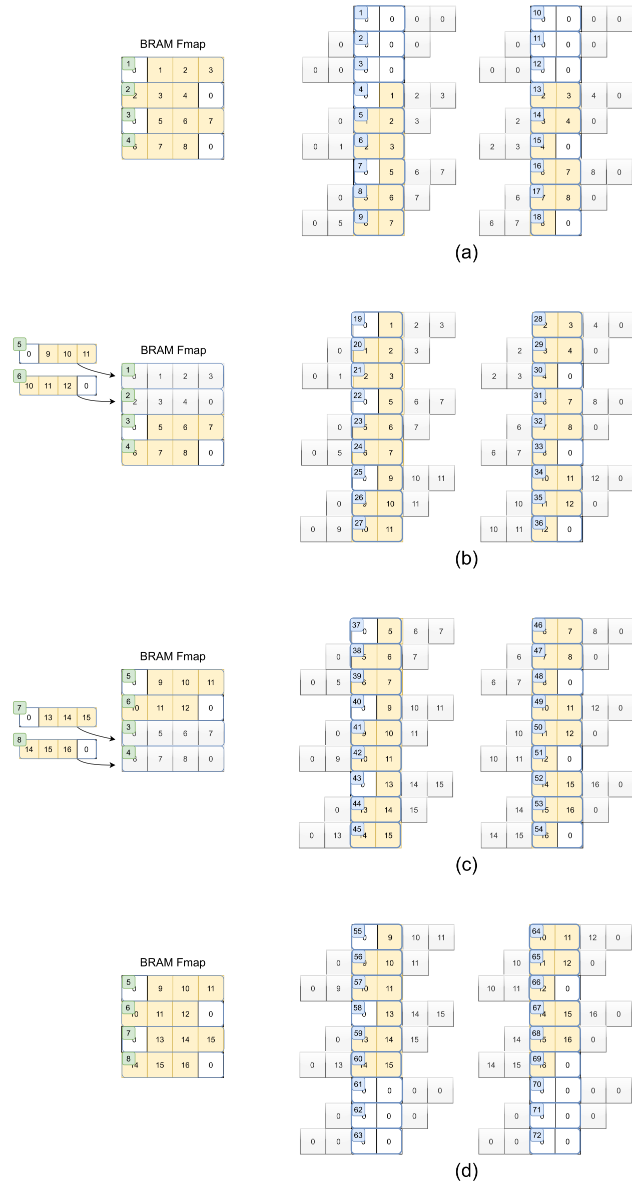


FIGURE 25. Process of FMAP matrix formation in FMAP matrix generation module.

memory utilization because we don't need to store all weight values as we will read them again in the repeating process.

2) WEIGHT ORDERING MODULE

Weight ordering module has similar functionality to mapping module in the input FMAP path. The task is to prepare the weight to a suitable data format, thus it will be easy for the next module (Weight Matrix Generator module) to produce the weight matrix. It also has the same architecture to mapping module, which can be seen on Figure 20. It consists of 3 registers to interchange the data.

Figure 26 shows the example of the input data and the output from weight ordering module with the time-stamp to show the order of data being received and produced. In the example, the input data has the width stream of 16 words and there are the total of 9 data packets. Each data packet has been labeled by the time-stamp which shows the order of data arrived (in case of input time-stamp) or the data produced (in case of output time-stamp). The output data has the width stream of 9 words and there are total of 16 data packets. To simplification, we can see this module as a stream width converter that converts the stream width of 16 to 9.

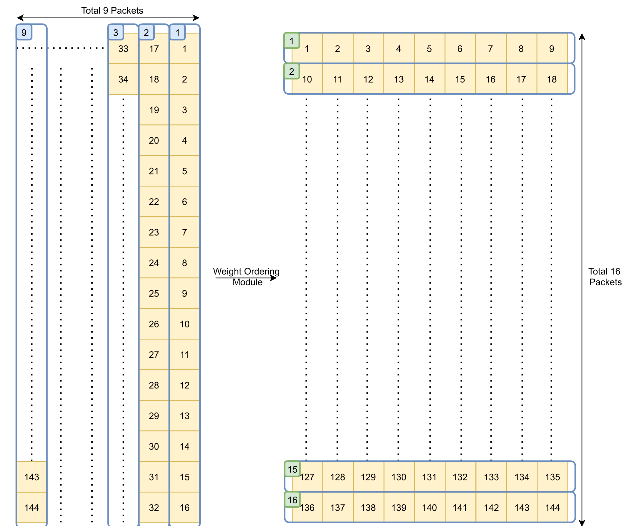


FIGURE 26. Input data and the output produced from weight ordering module.

The scheduling operation for this module is also similar to mapping module, as shown in Figure 21. The only difference is that the read operation is not always done in every clock cycle but sometimes in every 2 or 3 clock cycles. The read operation will be performed only when we need the data from the input to produce the output data. For example, as in Figure 26, the first output time-stamp requires only the data from the first input time-stamp, thus we perform the read operation to get the first input time-stamp. The second output time-stamp requires the part of first input time-stamp and also the other part of the second input time-stamp. Thus, we perform the read operation again to get the second input time-stamp. The third output time-stamp needs only the second input time-stamp which had been read before, thus the read operation won't be performed again until it needs the next input time-stamp to produce the next output time-stamp.

3) WEIGHT MATRIX GENERATOR MODULE

Weight matrix generator module main task is to produce the weight matrix $F[K][RS = 9]$. Figure 27 shows the internal architecture of Weight Matrix Generator module. It consists of register, BRAM weight to store temporary the value of weight that will be used in the near time, and mux to select the target of the read operation. There are 4 operations in this module. Read operation loads the data to RegIn registers from either the input FIFO or BRAM weight. Store operation and Write operation can execute in the same time.

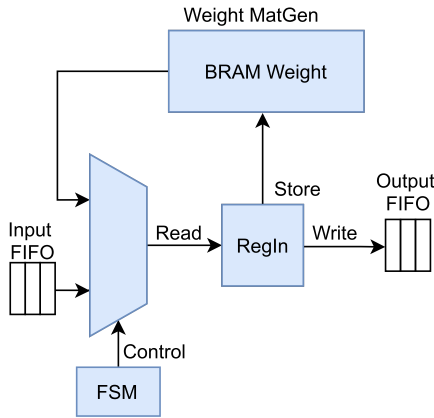


FIGURE 27. Architecture of weight matrix generator.

The Store operation stores the data in RegIn registers into the BRAM weight. And, the write operation will write the output to output FIFO. The Store operation will execute only when the new data packet from input FIFO has been read. Lastly, the Control operation will be handled by the FSM which schedules the next operation in every clock cycle and determines the target of the read operation.

There is no further data arrangement needed for this module as the matrix form is already formed from the last module. As shown in Figure 26, the right side of the figure produces the 16 packets of data which corresponds to the weight matrix with the size of $K \times 9$, where K equals to 16. The BRAM stores all K weights data in one channel only. When the new weight from new channel is stored, it will overwrite the old weight data. Depending on the choice of tiles size in block matrix multiplication, the same $K \times 9$ weight matrix needs to be produced multiple times. This is the reason why the weights values need to be store in the BRAM. Figure 28 shows how the operation will be scheduled. This module manages to achieve 1 clock cycle of initiation interval and 3 clock cycles latency.

Weight Matrix Generator (II=1, Latency 3)										
Data In (9 words width)	Clock Cycles									
	C0	C1	C2	C3	C4	C5	C6	C7	C8	C9
1st		Control	Read	Store/ Write						
2nd			Control	Read	Store/ Write					
3rd				Control	Read	Store/ Write				
					Control	Read	Write			
						Control	Read	Write		
							Control	Read	Write	
4th								Control	Read	Store/ Write

FIGURE 28. Operation scheduling in weight matrix generator module.

E. MATMULT CORE

Matmult core module consists of systolic arrays and BRAMs to store the partial sums during the computation. It basically implements the computation in Equation (1) with the dataflow from Algorithm 2. FMAP matrix size should be the multiple size of systolic array. For our case, we choose systolic array with the size of 9×14, because it is least common multiple size of FMAP matrix size across all batch layers. Figure 29 shows the example of how block matrix multiplication will be executed on MatMult Core. The order of computation starts from computing the matrix multiplication between matrix A and matrix D, then matrix A and matrix E. Both results of these matrix multiplication will be stored in partial sum BRAM. The next batch computation will be the multiplication between matrix B and matrix F, then matrix B and matrix G. The partial sum of this batch computation will be added to the partial sum of last batch computation (matrix multiplication between matrix AD and AE). It will be saved on the same location of the last partial sum, thus it won't require new memory space. The last batch computation is multiplication between matrix C and matrix H, then matrix C and I. The result will also be added to the last partial sum.

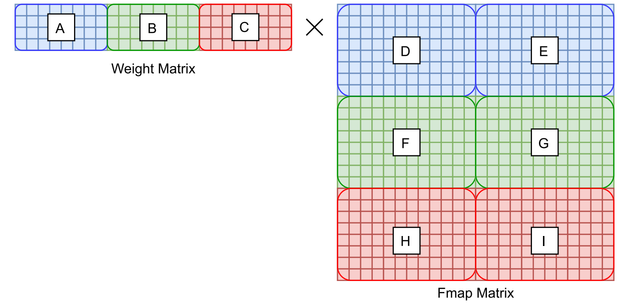


FIGURE 29. Block matrix multiplication order.

Figure 30 shows the internal architecture of MatMult core. Systolic array consists of 9×14 processing element. Each processing element consists of MAC (multiply-accumulate) operation and one FMAP register to store one pixel of FMAP. FMAP data from the FMAP matrix generator will be placed inside each processing element as stationary data. The moving data will be weight matrix and bias or partial sum matrix. One loop process of block matrix multiplication or addition can be summarized as follows:

- First, load the FMAP to the processing elements. It takes 9 clock cycles where in each clock cycle, 14 processing elements are filled at once.
- Then, compute the multiplication or the addition, while streaming weight and bias or partial sum matrix data. It takes $K + 9$ clock cycles depending on the number of output channels for each layer.

The idle time for matrix multiplication/addition in one loop process is only 9 clock cycles (during the FMAP filling process). It is worth mentioning that the bigger output channel K is, the more efficient our systolic arrays will be (idle time to computation time ratio will be smaller). It takes 9 clock cycles to execute 9 LoMap (Load Map) operations, then it

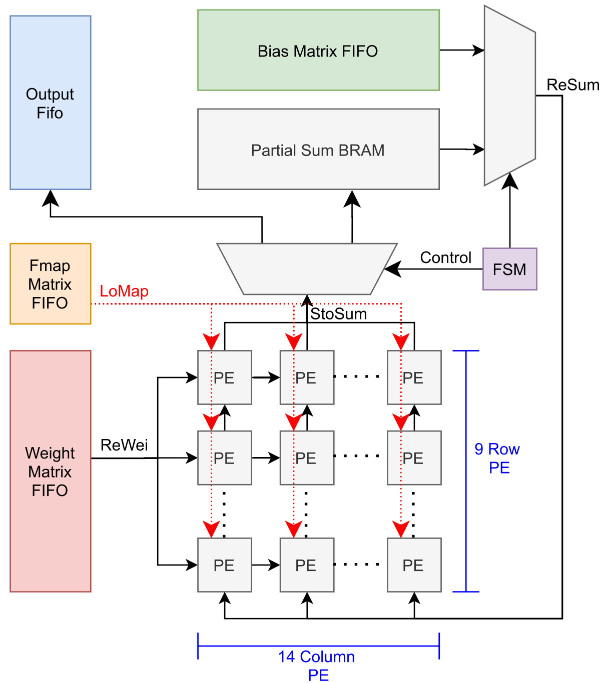


FIGURE 30. Architecture of Matmult core.

takes $9 + K$ clock cycles for the computation. In the case of multicore architecture, the computation will take $9 + \frac{K}{Ncore}$ clock cycles, where $Ncore$ is the number of MatMult Core. During computation process, several operations executed at the same time: (1) ReWei (Read Weight) operation to read weight matrix and to stream it to systolic array. (2) ReSum (Read Partial Sum) operation to read the last partial sum that will be added to the next partial sum (or use bias matrix for initial value). (3) StoSum (Store Partial Sum) operation to store the accumulated partial sum to the partial sum BRAM (or directly to output the FIFO when the final accumulated result is obtained).

F. OUTPUT FMAP PATH

Output FMAP path consists of DLQ module, MP module, Up Sampling module, Output Ordering module, and Output Writer module. DLQ module combines the functionality of Dequantization, leaky ReLU, and Quantization. Max pooling module handles the max pooling operation of each batch layer. Max pooling will reduce the size of the FMAP (depending on the number of stride), while up-sampling module will double the size of FMAP. Both max pooling and up-sampling modules can also be skipped depending on the parameter given by the instructions. Output ordering module prepares the data format, thus it will be easy for the output to be written into the DDR memory. Lastly, the output writer works in similar way to the other reader modules (Bias Reader, FMAP Reader, and Weight Reader). Instead of reading from theDDR, its task is to write back to the DDR. The details for each modules will be explained in following sections.

1) DLQ MODULE

DLQ module performs the Dequantization, leaky ReLU, and Quantization processes that have been explained in Section II-G. Figure 31 shows the internal architecture of DLQ module. It consists of 14 parallel multipliers to multiply the FMAP outputs of convolutions with the positive scales or negative scales. The output of this multiplication will become the quantized FMAP value in the form of 8-bit data. Figure 32 shows how the scheduling in DLQ module is performed. Because the scales are in fixed point numbers, the multiplication processes are also done in fixed points and it takes 4 clock cycles to finish a multiplication. Overall, this module has initiation interval of 1 clock cycle and a latency of 7 clock cycles.

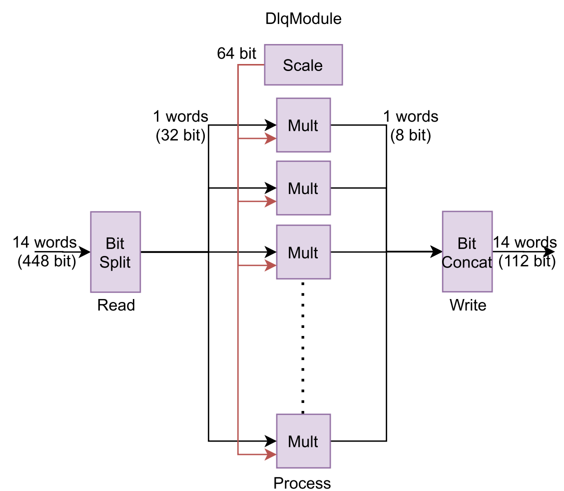


FIGURE 31. Architecture of DLQ module.

DLQ Module (II=1, Latency=7)													
Data In (448 bit) (14 words width)	Clock Cycles												Data Out (112 bit) (14 words width)
	C0	C1	C2	C3	C4	C5	C6	C7	C8	C9	C10	C11	
1st		Control	Read	Process				Write					1st
2nd			Control	Read	Process				Write				2nd
3rd				Control	Read	Process				Write			3rd
4th					Control	Read	Process				Write		4th
5th						Control	Read	Process				Write	5th

FIGURE 32. Operation scheduling in DLQ module.

2) MP MODULE

MP module performs max pooling operation to the output FMAP. It also can be skipped because not every batch layer needs max pooling process. Figure 33 shows the ordering of the FMAP output produced from the MatMult Core and the DLQ module. The producing order is based on the channel. In each channel, it produces 14 words width of data (1 data packet). Because of this, MP module needs to store the FMAP values before it can be compared to the other FMAP values in different row in the same channel.

Figure 34 shows the internal architecture of MP module. It consists of two comparators. The first comparator will compare the FMAP value in the same row. The second

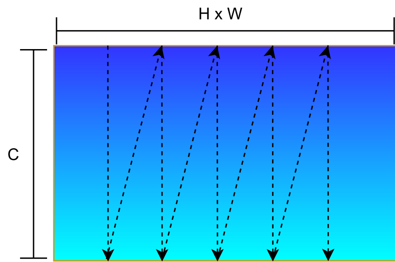


FIGURE 33. FMAP output producing order.

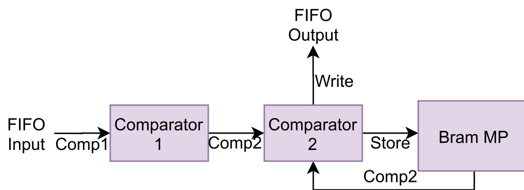


FIGURE 34. MP module internal architecture.

comparator will compare the FMAP value in different row. At the beginning, the BRAM MP is initialized with minimum values, thus the first row of the FMAP is always stored in the BRAM. Figure 35 shows an example how the FMAP data is being processed inside the module. In this example, we use the FMAP data packet with the width of 8 words. The output of the first comparator will be reduced the width of FMAP words to only 4 values (assuming MP size is 2 by 2 with the stride of 2). This output will be compared again using the second comparator with another 4 width FMAP words from different row (but still in the same channel), that has been stored in the BRAM. The final output of the MP module will be the max pooling result with the reduced width of data packet (in the example of Figure 35, the input has 8 words width and output of 4 words width). To preserve the words width, we will stored again the output words in another BRAM (not shown in the Figure) and will output the final results once the words width is the same as the input words width. In our real implementation, this module receives the data packets with the width of 14 words and produces the data packet with the width of 14 words.

Figure 36 shows how the operations are scheduled. This module has initiation interval of 1 clock cycle and a latency (loop latency) of 3 clock cycles. The overall output latency of this module will depend on the size of the FMAP being processed. Specifically, it depends on the number of data packets in each FMAP row. For example, the FMAP with the size of 14 by 14 only has 1 data packets to read the data in one row, because the width of our data packet is equal with the number of data in one row. However, FMAP with the size of 28 by 28 has 2 data packets to read one row of FMAP. As a result, it needs to store more data packet before it can be compared to the next row in the same channel.

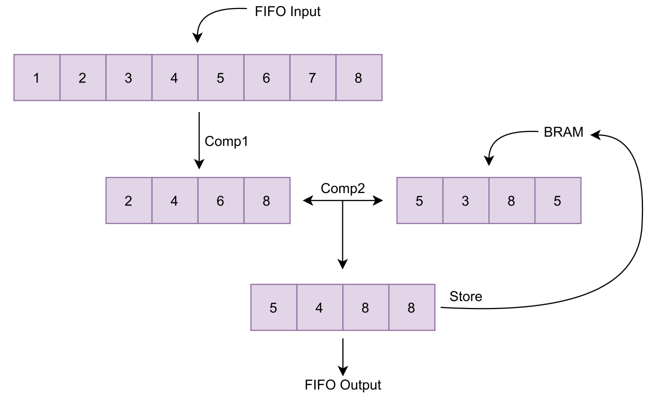


FIGURE 35. The working illustration of MP module.

MpModule (II=1, Latency 3)												
Data In (14 words width)	Clock Cycles											Data Out (7 words width)
	C0	C1	C2	C3	C4	C5	C6	C7	C8	C9	C10	
1st		Comp1	Comp2	Store								
2nd			Comp1	Comp2	Store							
3rd				Comp1	Comp2	Store						
4th					Comp1	Comp2	Store					
5th						Comp1	Comp2	Write				1st
6th							Comp1	Comp2	Write			2nd
7th								Comp1	Comp2	Write		3rd
8th									Comp1	Comp2	Write	4th

FIGURE 36. Operation scheduling in MP module.

3) UP SAMPLING MODULE

Up-sampling module performs up sampling operation to increase the FMAP size by duplicating each pixel area by 4. It also can be skipped because not every batch layer needs up sampling operation. Figure 37 shows the internal architecture of up-sampling module. It consists of 3 registers and a BRAM to store the up-sampling results. The RegIn registers store the data packet of 14 words (1 data packet) during the Read operation. The Dupli operation will duplicate the data in the RegIn registers to the Duplicator registers which stores the data packet of 28 words. This operation performs the up-sampling operation for the FMAP data in the same row. As a result, the data packet width will be doubled. The result of Duplicator registers will be stored in BRAM UpSampling, while the first 14 words of data packet will be loaded to the RegOut registers during the Load operation. Lastly, the Write operation will write out the results to the output FIFO. Once one full FMAP row has been produced, this module will produce the same outputs again using the previous values that had been saved on BRAM, where the up-sampling operation performs the pixel area duplication by 4. The first 2 duplications are produced in the first row, the last 2 duplications are produced in the next row which essentially the same row as before. During this phase, the Read, the Dupli, and the Store operations won't be performed. It will just load the previous values from the BRAM.

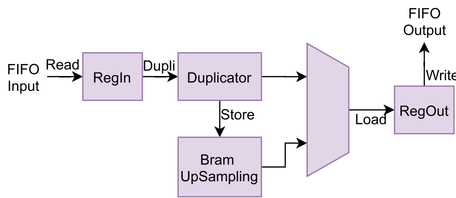


FIGURE 37. Up sampling module internal architecture.

Up Sampling Module (II=1, Latency 4)												
Data In (14 words width)	Clock Cycles											Data Out (14 words width)
	C0	C1	C2	C3	C4	C5	C6	C7	C8	C9	C10	
1st		Read	Dupli	Store/ Load	Write							1st
2nd			Read	Dupli	Store/ Load	Write						2nd
3rd				Read	Dupli	Store/ Load	Write					3rd
4th					Read	Dupli	Store/ Load	Write				4th
								Load	Write			5th
									Load	Write		6th
										Load	Write	7th

FIGURE 38. Operation scheduling in up sampling module.

Figure 38 shows how operations are scheduled in this module. The operation started by the Read operation following by the Dupli operation. The Store and the Load operations can be executed on the same time. There are two possible sources for the Load operation. The first Load operation is done during the initial phase, where the source of Load operation is from the Duplicator registers. The second load operation is done when the Read, the Dupli, and the Store operations are not performed and the source of the load operation is from the BRAM Up Sampling. This module manages to achieve initiation interval of 1 clock cycle and the maximum latency of 4 clock cycles.

4) OUTPUT ORDERING MODULE

Output ordering module performs the stream width conversion from 14 words width to 16 words width. The architecture of this module is also similar to the Mapping Module in the Input FMAP Path and the Weight Ordering Module (Figure 20). It consists of memories variables (for this module, we use BRAM) to exchange data words and the final output stream has the width of 16. The reason why we are doing this is because of the AXI Full protocol specification [18]. It requires to have the output stream width of 16 before it can write the result back to the DDR. The operation scheduling for this module is also similar to Figure 21. It has initiation interval of 1 clock cycle and a latency (loop latency) of 3 clock cycles.

5) OUTPUT WRITER

Output writer module performs similar task to the other reader modules. However, instead of reading from the DDR memory, this module will write the result to the DDR memory through AXI Full protocol. The internal architecture of this module also similar to Figure 11 but with backward arrow (indicating the write operation to DDR memory). The operation scheduling also similar to Figure 12. It has initiation interval of 1 clock cycles and latency of 3 clock cycles.

V. PROCESSING FLOW

This section discusses how the processing flow between ARM host CPU and FPGA chip will be executed. Figure 39 shows the working flow between ARM processor as the host CPU and the FPGA chip as the accelerator. ARM processor initiates the first process by setting up accelerator's parameters and then triggers the start signal to the accelerator. The accelerator will execute all the instructions given to it (1 instruction 1 batch layer) until it trigger back finish signal to the ARM processor. After that, ARM processor reads the result and applies it to the image to form the boundary box and the classification results.

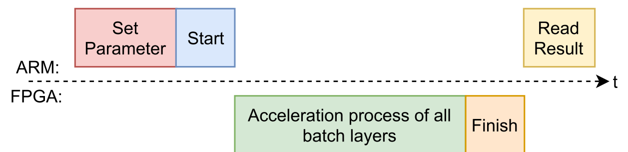


FIGURE 39. Processing flow between ARM host CPU and FPGA chip.

The details of acceleration process inside the FPGA are shown in Figure 40. The process is started by loading all the instructions from the DDR memory into the BRAM instruction memory. Once all instructions have been loaded, the first instruction will be fetched from the BRAM and then will be decoded to get the necessary parameters for accelerator. Once the acceleration process for one batch layer is started, it will then execute in parallel and pipeline fashion until all the outputs have been written back to the DDR. After that, the process of fetching and decoding for the next batch layer will be started. This process keeps repeating until all instructions (all batch layers) have been executed.

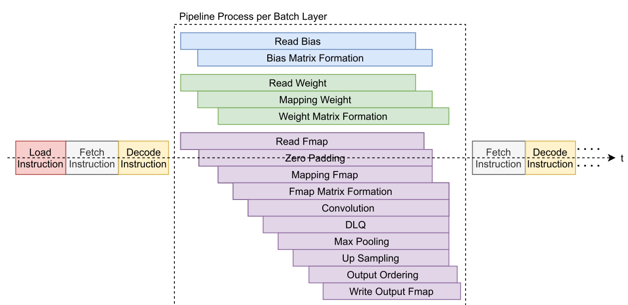


FIGURE 40. The processing flow Inside FPGA.

VI. CUSTOM INSTRUCTION SET

This section discusses the implementation of custom instruction set for our hardware accelerator. The instruction is

already prepared in the DDR memory and the user just needs to pass the base address along with the number of instructions through AXI Lite interface before triggering the start signal to the accelerator. Figure 41 shows our custom instruction fields. The total length of the instruction is 512 bits. The first 4 fields, where each field has length of 32 bits, are the base address for bias, input FMAP, weight, and output FMAP for each batch layer. The next 2 fields, where each field has the length of 32 bits, are the positive scale and negative scale values which will be used by the DLQ module. The last field with the length of 320 bits consists of several parameters for each batch layers.

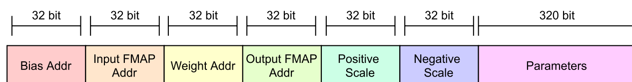


FIGURE 41. Custom instruction fields.

TABLE 2. Details of parameters field in custom instructions.

Layer	Parameters							
	FMAP Dim	Kernel Dim	Input Channel	Output Channel	Zero Pad	Max Pooling	Up Sampling	Number of Tiles
1	448	3	3	16	1	2	0	56
2	224	3	16	32	1	2	0	28
3	112	3	32	64	1	2	0	14
4	56	3	64	128	1	2	0	7
5	28	3	128	256	1	2	0	7
6	14	3	256	512	1	1	0	2
7	14	3	512	1024	1	0	0	7
8	14	1	1024	256	0	0	0	1
9	14	1	256	128	0	0	1	1
10	28	3	384	256	1	0	0	14
11	28	1	256	18	0	0	0	1
12	14	3	256	512	1	0	0	2
13	14	1	512	18	0	0	0	1

Table 2 shows some of the parameters in the parameter fields. Note that not all parameter is presented in table 2. This is the simplification of some parameters. The other parameters consist of several loop parameters which indicate the number of executions for each module in the accelerator. Some other loop parameters are derived during the instruction decoding process. Table 2 also shows how the parameters change across all 13 batch layers. The FMAP Dim field contains the input FMAP size for each batch layer. The Kernel Dim field contains the size of kernel used during convolution layer. The Input and the Output Channel field contain the number of the input channels and the output channels for each batch layer. The Zero Pad field only has 2 possible values. The value 1 means that zero padding process will be executed and the value 0 means that zero padding process will be skipped. The Up Sampling field contains the stride number of 2 by 2 max pooling window. If the number is zero, it means the up sampling process will be skipped for that particular batch layer. The Number of Tiles field contains the number of block matrix multiplications will be processed in one loop of computation before the output will be produced. All values of these parameters are chosen to optimized the BRAM memory space limitation in our FPGA.

VII. EXPERIMENTAL SETUP

This section discusses the experimental setup that we used for testing our design. Figure 42 shows how our system is

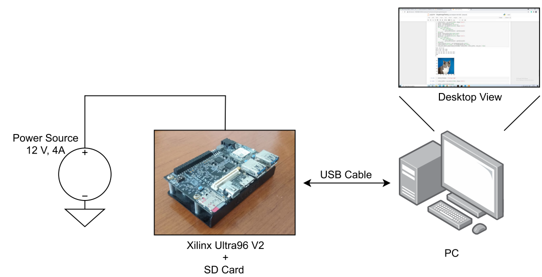


FIGURE 42. Experimental setup diagram.

organized. We use Xilinx Ultra96 V2 as our FPGA platform. It is powered through 12 V voltage and 4 A current in DC power source. We also use SD card which includes the image for testing purpose. To access the FPGA, we use the PYNQ as hardware abstraction layer. PYNQ is already installed in the SD card and we can access the FPGA directly when it is connected to the PC through USB cable.

We do the single image testing for detecting whether the object (*e.g.,* the cat) exists in some images or not. Figure 43 shows the live system setup when we tested our design. The FPGA board is shown in bottom left corner of the image, it is connected to the computer through USB cable.

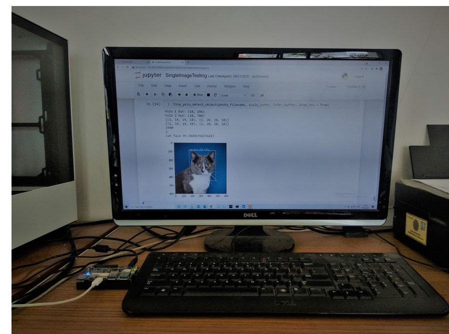


FIGURE 43. Live experimental setup.

VIII. EVALUATION

This section discusses analysis and comparison of our accelerator. In Subsection 44, we analyze the dataflow and workload impact on the memory access. In Subsection VIII-B, we discuss the impact of our *im*2*col*() function implementation on the memory storage. In Subsections VIII-C and VIII-D, we give the comparison of our accelerator with other hardware platform (CPU, GPU, & ASIC) and other FPGA hardware accelerator respectively.

A. NUMBER OF MEMORY ACCESS

In this section, we will derive the number of memory access in each level of memory hierarchy. The number of memory access can be used to estimate the energy consumption of the accelerator (the higher the memory level in hierarchy, the more expensive energy required to access the data). At least the energy consumption from the data movement, it is

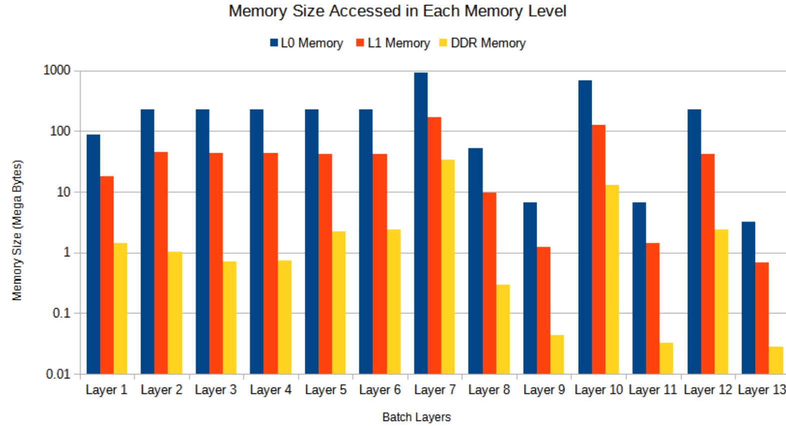


FIGURE 44. L0, L1, and DDR memory access per batch layer (computed from Equation (14)).

usually the most dominant energy consumption. Rather than giving an estimation of energy consumption in physical unit, we derive the analytical model for our accelerator based on the number of memory access [21]. By this way, we can use this model even if we implement our accelerator in different platforms.

The model is derived based on Algorithm 2. There are 3 data types which will be our concern: IFMAP (input FMAP), OFMAP (output FMAP), and Weight. We neglect the bias data movement, because the number of bias is small compared to other data types. For each of memory level, the number of memory access is as follows:

$$\begin{aligned} DDR_N &= DDR_I + DDR_W + DDR_O \\ L1_N &= L1_I + L1_W + L1_O \\ L0_N &= L0_I, \end{aligned} \tag{14}$$

where:

DDR_N = Total DDR memory access,
$L1_N$ = Total L1 memory access,
$L0_N$ = Total L0 memory access,
DDR_I = IFMAP access on DDR memory,
$L1_I$ = IFMAP access on L1 memory,
$L0_I$ = IFMAP access on L0 memory,
DDR_W = Weight access on DDR memory,
$L1_W$ = Weight access on L1 memory,
DDR_O = OFMAP access on DDR memory, and
$L1_O$ = OFMAP access on L1 memory.

Start from the DDR memory level, the value of DDR_I, DDR_W, and DDR_O can be derived from the number of *LoadFmap*(), *LoadWeight*(), and *WriteFmap*() function calls, respectively, as shown in Algorithm 2 lines 11, 13, and 31 that are multiplied by the number of byte data accessed in one function calls as follows:

$$\begin{aligned} DDR_I &= NH1 \times DRbyte_I \\ DDR_W &= NH1 \times C \times DRbyte_W \\ DDR_O &= NH1 \times DRbyte_O, \end{aligned} \tag{15}$$

where *DRbyte* is the number of byte accessed in one function call for respective data types. The value of *DRbyte* differs across batch layers. It is because the dimensions of FMAP and filter are also vary.

In L1 memory level, the value of $L1_I$ can be derived from the number of *Fill*() function calls in Algorithm 2 (line 16). This function takes a set of values from L1 memory and expands the values according to the padding and *im2col*() function applied in each respective layer. Then, it stores the value in the L0 memory. The value of $L1_W$ and $L1_O$ can be derived from the number of L1 memory accessed during the computing process (line 23) as follows:

$$\begin{aligned} L1_I &= NH1 \times C \times H1 \times NW1 \times L1byte_I \\ L1_W &= NH1 \times C \times H1 \times NW1 \times K \times L1byte_W \\ L1_O &= NH1 \times C \times H1 \times NW1 \times K \times L1byte_O, \end{aligned} \tag{16}$$

where $L1byte$ is the number of byte accessed in one function call or the number of data read from the L1 memory level for respective data type. For $L1byte_I$, the value will be different depending on the use of *im2col*() function to expand the IFMAP tile. Note that some batch layers use 1 by 1 filter. In that case, *im2col*() function is not required. There are 2 possible values for $L1byte_I$: 3×16 (if *im2col*() is used) or 9×14 (if *im2col*() is not used). The values of $L1byte_W$ and $L1byte_O$ depend on the systolic array size. For our implementation, the value will be 9×14.

Finally, the value of $L0_I$ can be derived the same way as $L1_W$ and $L1_O$.

$$L0_I = NH1 \times C \times H1 \times NW1 \times K \times L0byte_I, \tag{17}$$

where $L0byte_I$ has the same value as $L1byte_W$ and $L1byte_O$.

Using this model, we can compute the memory access size on each memory level. Figure 44 shows the memory access size for all batch layers. As shown, the L0 memory level dominates the memory size access across all batch

layers, followed by the L1 memory and the DDR memory, respectively. This result is to be expected, since we want to minimize the memory access from the DDR and optimize the memory access in the L0 memory level. The high number of memory access to/from the L0 and the L1 memories also indicates the high number of data reuse.

From this model, we can further estimated the energy cost from the data movement by multiplying the result with the amount of energy used per byte of data movement (joule/byte) for each memory level in the targeted device.

B. ANALYSIS ON Im2col

In this section, we will evaluate the *im2col*() function implementation in our hardware, especially in the term of memory size. As we know, the naive implementation of *im2col*() function will expand the IFMAP size from the dimension of $I[C][H][W]$ to $I[CRS][PQ]$. In our YOLO model, we always use the same type of convolution so $PQ == HW$ and our IFMAP size will expand by the RS times. Except for layer 8, 9, 11, and 13 (which uses 1 by 1 filter), we use 3 by 3 filter for convolution, thus the IFMAP expansion ratio is 9.

Performing the naive *im2col*() function directly to IFMAP will result 9 times size of expansion. To prevent this, instead of directly transform IFMAP using *im2col*(), we transform the IFMAP into compressed matrix form and store it to the L1 memory. In general, the size IFMAP size expansion is as follows:

$$\frac{TransformedSize}{OriginalSize} = \frac{H \times C \times PacketSize \times \frac{W}{W1}}{W \times H \times C} = \frac{PacketSize}{W1}. \quad (18)$$

As mentioned in Section IV-C, for our case, the $PakcetSize = 16$ and $W1 = 14$, thus the expansion ratio is $\frac{8}{7} \approx 1.14$. Therefore, we can save storage space up to ≈ 7.89 compared to the naive *im2col*() implementation. Figure 45 shows the bar chart of *im2col*() size expansion when it is implemented naively vs using the compressed form.

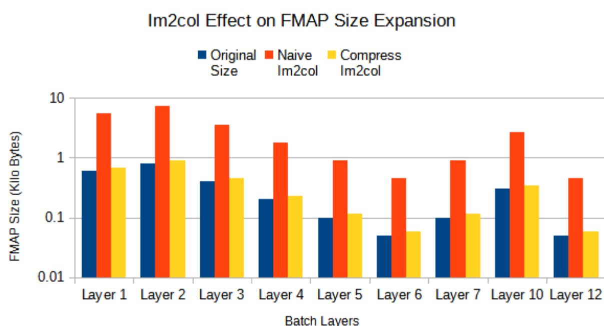


FIGURE 45. FMAP size expansion caused by *im2col*() function (Compressed vs Naive).

C. COMPARISON WITH CPU, GPU, & ASIC

The accelerator is implemented in Ultra96 V2 platform board. It manages to run with the maximum clock frequency of 250MHz and has a power consumption of 4.3 W. The design takes 248 BRAMs, 242 DSPs, 27.3k LUTs, and 38.5k FFs. The total latency for 1 image with the image size of 448 × 448, that going through 13 batch layers, is 121ms. The theoretical peak GOPS for current version of design is 31.5 GOPS and has a power efficiency of 7.4 GOPS/W.

Table 3 shows layer by layer performance comparison between Ryzen 5 3600 CPU and our accelerator implementation on FPGA. To be fair, we compare only convolution time in CPU (which is only includes matrix multiplication and is done by using standard python NumPy library) with the overall execution time in our accelerator. The data preparation time and post-processing time are not included in CPU execution time because there is no standard library that can guarantee the quality of code. Thus, the code that implements the data preparation and the post-processing in CPU will be different from person to person. By comparing the result with the same YOLO model that runs on AMD Ryzen 5 3600 CPU with 3.6 GHz clock speed, our accelerator is 69.3 times faster.

TABLE 3. Performance comparison to Ryzen 5 3600 CPU.

Batch	MAC	CPU		FPGA	
Layer	Operation	Conv Time	GOPS	Exec Time	GOPS
1	86,704,128	1149 ms	0.08	5.84 ms	14.85
2	231,211,008	2995 ms	0.08	11.47 ms	20.16
3	231,211,008	386 ms	0.6	9.40 ms	24.58
4	231,211,008	146 ms	1.58	8.38 ms	27.58
5	231,211,008	139 ms	1.66	7.94 ms	29.13
6	231,211,008	141 ms	1.64	7.64 ms	30.26
7	924,844,032	1580 ms	0.59	29.96 ms	30.87
8	51,380,224	84 ms	0.61	1.77 ms	29.00
9	6,422,528	8 ms	0.80	0.30 ms	21.13
10	693,633,024	1025 ms	0.68	23.64 ms	29.34
11	3,612,672	3 ms	1.20	0.33 ms	10.88
12	231,211,008	279 ms	0.83	7.60 ms	30.42
13	1,806,336	2 ms	0.9	0.16 ms	11.01

We also compare our accelerators with commercial AI accelerators on the market that use different types of platforms. First, we compare it to a GPU-based AI accelerator, the Jetson Nano. Jetson Nano is a GPU-based hardware accelerator for edge computing purposes [22]. Jetson Nano achieves quite impressive inference performance, which only requires 40 ms execution time for the YOLOv3-Tiny algorithm using 32-bit floating point precision. The GPU on Jetson Nano has been optimized so that it can work at a lower clock frequency than the GPU on a PC. Even so, the clock frequency still reaches 1.3 GHz and also requires a fairly high power consumption of up to 10 Watts. When compared to our accelerators, our power consumption is 2.34× lower than the Jetson Nano.

We also compare with another type of accelerator which is an ASIC-based accelerator. The name is Intel's Neural Computing Stick (NCS) 2 [23]. In terms of platform, ASIC implementation of the Intel's accelerator is indeed the most optimal in terms of energy efficiency and computational performance.

NCS has a clock frequency of 700 MHz and is capable of performing computations with a precision of 16-bit floating point. We tried running the YOLOv3-Tiny algorithm on NCS and got an inference execution time of 55 ms. Compared to the Jetson Nano, the execution time is still inferior. However, in terms of power consumption, the NCS only requires 1 Watt, which means it is very power-efficient.

In Table 4, we compare the performance of AI accelerators from 3 different hardware platforms. Comparing in terms of power consumption is certainly unfair because accelerators made using ASIC can certainly save further energy, while GPU-based accelerators are definitely the most energy-intensive. Meanwhile, when compared in terms of the overall inference execution time, GPU-based accelerators still have the advantage. Currently, GPU-based accelerators are still the *de facto* standard in the industry because they have high computing performance and have the most complete design tools for developers. One of the things that causes GPUs to have high computing performance is because GPUs have a high level of parallelization and can run at high clock frequencies as well.

TABLE 4. Performance comparison to GPU and ASIC based AI accelerator.

	Movidius NCS 2	Jetson Nano	Our Work 1 Core*	Our Work 8 Core**
Platform Type	ASIC	GPU	FPGA	FPGA
Clock Freq	700 MHz	1300 MHz	250 MHz	214 MHz
Bit Precision	FP16	FP32	INT8	INT8
Latency	55 ms	40 ms	121 ms	52 ms
Total Clock Cycle	38.5 M	52 M	30.2 M	11,1 M
Clock Cycle Ratio	1.28×	1.72×	1×	0.36×
Power	1 Watt	10 Watt	4.26 Watt	5.44 Watt

* Implemented on Xilinx Ultra96 V2 FPGA, ** Implemented on Xilinx ZCU104 FPGA

For fairer comparison, we decided to ignore performance in terms of power consumption because it really depends on the choice of a platform. As shown in Table 4, we compare the performance in terms of architecture by calculating the number of clock cycles required in one inference process. A better architecture means it can complete computations with fewer number of clock cycles. In addition, by comparing the number of clock cycles, we can compare the computational performance of accelerators more fairly even though their base clock frequencies differ due to the influence of platform choice. From the number of clock cycles required for one inference process, the accelerator that we built using the FPGA platform has the least number of clock cycles. This shows that in terms of architecture, our accelerator is more optimal. In addition, we can also increase the number of compute cores in our accelerator from 1 core to 8 cores. As a result, the execution time of our accelerator dropped to 52 ms, less than the execution time on NCS which runs at a higher clock frequency.

D. COMPARISON WITH OTHER WORKS

Table 5 shows the comparison between this work and other works. The results in this table are mostly obtained from the design tools we use, namely Xilinx Vivado 2020.1 [24]. Latency is calculated by measuring the average image inference time performed using the python standard time library running on PS side. The GOPS value listed is the theoretical peak GOPS value obtained from the following equation [3]:

$$\frac{operation}{second} = \left(\frac{1}{\frac{cycles}{operation}} \times \frac{cycles}{second} \right) \times \text{number of PEs} \times \text{utilization of PEs}, \quad (19)$$

where $\frac{cycles}{operation}$ indicates the number of clock cycle needed for 1 MAC operation (in our case, our systolic array only needs 1 cycle per MAC operation), $\frac{cycles}{second}$ indicates the clock frequency, number of PE indicates the number of available PE on accelerator (in our case, 1 MatMult core consists of 9×14 PE), and utilization of PE indicates the number of PE that is actually used for computation. The utilization of PE depends on the mapping strategy used, the model being executed, and the scheduling strategy. Since we calculated the peak value of GOPS, we assumed that PE utilization was 100%. However, in reality, the PE utilization value will vary from layer to layer.

TABLE 5. Performance comparison to other accelerators.

	[16]	[13]	[12]	this work
Year	2019	2020	2020	2021
Target Network	YOLO v2-Tiny	YOLO v3-Tiny	YOLO v3-Tiny	YOLO v3-Tiny
Precision	6b	16b	18b	8b
Clk Freq	200 MHz	100 MHz	200 MHz	250 MHz
Platform	Virtex-7 VC707	ZedBoard	Virtex-7 VC707	Ultra96 V2
Platform Price	$3,495	$475	$3,495	$249
BRAM	1026	185	141	248
DSP	168	160	2304	242
LUT	86k	25.9k	48.5k	27.3k
FF	60k	46.7k	93.2k	38.5k
Gate Counts Conversion	936k	482k	934k	433k
Latency	9.15 ms	532 ms	-	121 ms
Tested Dataset	Pascal VOC 2007	-	-	Custom VOC 2007
mAP	64.16%	-	-	75.00%
mAP Drop	2.63%	-	-	2.00%
GOPS	464.70	10.45	460.80	31.50
Power	18.29 W	3.36 W	4.81 W	4.26 W
GOPS/W	25.41	3.11	95.80	7.40

Nguyen *et al.* [16] managed to achieve impressive latency result for YOLO v2-Tiny accelerator by developing fully pipeline architecture across the different batch layers. This can be achieved by actually implementing entire batch layers in FPGA and embedding the weight directly to the FPGA. Thus, it reduces the bandwidth requirement for transferring weight values from the external DDR memory to the FPGA. The trade-off for this implementation requires a lot of resources. They managed to suppress this drawback a little

by reducing the precision up to 6 bits and by using binarized (one bit) weight value. In the end, this will reduce the classification accuracy while in our case, we maintain precision of 8 bits for both FMAP and weight. Even after the resources optimization, especially the BRAM and the LUT usages, their designs are still relatively high and not suitable for low-cost FPGA. Another drawback from directly embedding the weight to the FPGA is also lack of hardware flexibility and reprogrammability, which become the problems when we need to update the weight values or changing the type of network. In contrary, our hardware accelerator provides its own custom instruction set which means our accelerator can be reprogrammed to accelerate various types of network.

Our convolution method using GEMM has more flexibility (generalization) and higher efficiency compared to the previous works. For example by Yu and Bouganis, the computation is done per kernel per channel [13]. In addition, our accelerator does not need to be reinitiated by the host CPU for every batch layer. It results in lower overall latency. Our proposed design which is also to be targeted on low-cost FPGA, manages to work in higher clock frequency indicating better worst path delay. We also manage to get GOPS improvement up to 3×, 2.4× improvement in power efficiency, and 4× better latency with only 1.5× increment in the DSP usage. Moreover, in-memory computing (IMC) can be adapted for further works to reduce the off-chip memory access [25].

The works by Ahmad *et al.* [12] builds YOLOv3-Tiny accelerator in high performance FPGA. By using a lot of DSPs, they managed to get impressive GOPS performance and relatively low power consumption. Their accelerator is focused to accelerate the convolution process only. The pre-processing (zero padding) and post-processing (activation function, max pooling, up sampling, etc) are done sequentially by using the soft-core microprocessor Microblaze. The sequential process of both the pre-processing and the post-processing should affecting the overall latency performance too. There is no report about their design latency but comparing to our design which done pretty much everything in FPGA, the overall latency from their design should be relatively high.

We also mention the comparison of platform cost that is needed to implement the design. Compared to the other works, we manage to implement our design on the least expensive FPGA, indicating that our design is suitable to be implemented on lost-cost FPGA for edge computing application. To elaborate more, we also mention the comparison of the Gate Counts which give rough estimate of our design size when implemented using ASIC. The estimate gate count value is derived based on the FPGA resource consumption [19] (roughly an FF equals to 7 gate counts and an LUT equals to 6 gate counts). Compared to other works, our design achieves the smallest gate counts numbers indicating that our design will be smaller when implemented using ASIC.

IX. CONCLUSION

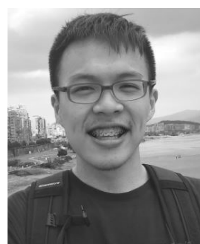
This paper presented a comprehensive hardware accelerator architecture for YOLOv3-Tiny implemented on low-cost FPGA Ultra96 V2. The YOLO algorithm was modified to optimize the hardware implementation. The modification included (1) reducing the precision up to 8 bits, while still maintaining the mAP of 75%, (2) merging the batch normalization operation with convolution operation, and (3) merging the dequantization, leaky ReLU, and quantization operation into a DLQ operation. The accelerator was also built by using GEMM principle for convolution. The proposed system managed to achieve the frame rate of 8.3 FPS with the throughput of 31.5 GOPS. We also introduced our own custom instruction set which increases hardware flexibility and reprogrammability when it comes to be implemented in other types of networks. Based on evaluation, our hardware accelerator managed to outperform the same task up to 69.3× faster compare to AMD Ryzen 5 3600 CPU. Moreover, compared to the other works, our accelerator has relatively small design size and small latency, where the clock cycle ratio of the proposed design is less 1.28× and 1.75× than that of the NCS 2 and the Jetson Nano, respectively. The architecture is useful and suitable for edge computing applications.

REFERENCES

[1] W. Shi and S. Dustdar, "The promise of edge computing," *Computer*, vol. 49, no. 5, pp. 78–81, 2016.
[2] X. Wang, Y. Han, V. C. M. Leung, D. Niyato, X. Yan, and X. Chen, "Convergence of edge computing and deep learning: A comprehensive survey," 2019, *arXiv:1907.08349*. [Online]. Available: http://arxiv.org/abs/1907.08349
[3] V. Sze, Y. Chen, T.-J. Yang, and J. S. Elmer, *Efficient Processing Deep Neural Networing*. San Rafael, CA, USA: Morgan et Claypool Publishers, 2020.
[4] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet classification with deep convolutional neural networks," *Commun. ACM*, vol. 60, no. 2, pp. 84–90, Jun. 2012.
[5] R. Girshick, J. Donahue, T. Darrell, and J. Malik, "Rich feature hierarchies for accurate object detection and semantic segmentation," *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, Jun. 2014, pp. 580–587.
[6] R. Girshick, "Fast R-CNN," in *Proc. IEEE Int. Conf. Comput. Vis. (ICCV)*, Dec. 2015, pp. 1440–1448.
[7] S. Ren, K. He, R. Girshick, and J. Sun, "Faster R-CNN: Towards real-time object detection with region proposal networks," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 39, no. 6, pp. 1137–1149, Jun. 2017.
[8] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi, "You only look once: Unified, real-time object detection," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit. (CVPR)*, Jun. 2016, pp. 779–788.
[9] J. Redmon and A. Farhadi, "YOLOv3: An incremental improvement," 2018, *arXiv:1804.02767*. [Online]. Available: http://arxiv.org/abs/1804.02767
[10] D. H. Jones, A. Powell, C.-S. Bouganis, and P. Y. K. Cheung, "GPU versus FPGA for high productivity computing," in *Proc. Int. Conf. Field Program. Log. Appl.*, Aug. 2010, pp. 119–124.
[11] M. Qasaimeh, K. Denolf, J. Lo, K. Vissers, J. Zambreno, and P. H. Jones, "Comparing energy efficiency of CPU, GPU and FPGA implementations for vision kernels," *Proc. IEEE Int. Conf. Embedded Softw. Syst. (ICESS)*, Dec. 2019, pp. 1–8.
[12] A. Ahmad, M. A. Pasha, and G. J. Raza, "Accelerating tiny YOLOv3 using FPGA-based hardware/software co-design," in *Proc. IEEE Int. Symp. Circuits Syst. (ISCAS)*, Oct. 2020, pp. 1–5.
[13] Z. Yu and C.-S. Bouganis, "A Parameterisable FPGA-tailored architecture for YOLOv3-tiny," *Appl. Reconfigurable Comput. Archit., Tools, Appl.*, vol. 5, pp. 330–344, Oct. 2020.
[14] M. Dukhan, "The indirect convolution algorithm," 2019, *arXiv:1907.02129*. [Online]. Available: http://arxiv.org/abs/1907.02129

[15] N. P. Jouppi, ''In-datacenter performance analysis of a tensor processing unit,'' in *Proc. 44th Annu. Int. Symp. Comput. Archit.*, 2017, pp. 1–12.
[16] D. T. Nguyen, T. N. Nguyen, H. Kim, and H. J. Lee, ''A high-throughput and power-efficient FPGA implementation of YOLO CNN for object detection,'' *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 27, no. 8, pp. 1861–1873, Aug. 2019.
[17] A. Ahmad and M. A. Pasha, ''Optimizing hardware accelerated general matrix-matrix multiplication for CNNs on FPGAs,'' *IEEE Trans. Circuits Syst. II: Exp. Briefs*, vol. 67, no. 11, pp. 2692–2696, Nov. 2020.
[18] *AXI Reference Guide*, Xilinx, San Jose, CA, USA, 2017.
[19] *Xilinx XAPP059 Gate Count Capacity Metrics for FPGAs*, Xilinx, San Jose, CA, USA, 1997.
[20] B. Jacob, S. Kligys, B. Chen, M. Zhu, M. Tang, A. Howard, H. Adam, and D. Kalenichenko, ''Quantization and training of neural networks for efficient integer-arithmetic-only inference,'' in *Proc. IEEE/CVF Conf. Comput. Vis. Pattern Recognit.*, Jun. 2018, pp. 2704–2713.
[21] A. Putra, T. Adiono, N. Sutisna, I. Syafalni, and R. Mulyawan, ''Hardware dataflow for convolutional neural network accelerator,'' in *Proc. Int. Symp. Electron. Smart Devices (ISESD)*, Jun. 2021, pp. 1–6.
[22] *Jetson Nano Data Sheet*, Nvidia, Santa Clara, CA, USA, 2020.
[23] *Intel Neural Compute Stick 2 Data Sheet*, Intel, Mountain View, CA, USA, 2019.
[24] *Vivado Design Suite User Guide*, Xilinx, San Jose, CA, USA, 2020.
[25] A. Sebastian, ''Memory devices and applications for in-memory computing,'' *Nat. Nanotechnol.*, vol. 15, pp. 529–544, Jul. 2020.

TRIO ADIONO (Member, IEEE) received the B.Eng. degree in electrical engineering and the M.Eng. degree in microelectronics from Institut Teknologi Bandung, Indonesia, in 1994 and 1996, respectively, and the Ph.D. degree in VLSI design from the Tokyo Institute of Technology, Japan, in 2002. He is currently a Professor at the School of Electrical Engineering and Informatics and also serves as the Head for the IC Design Laboratory, Microelectronics Center, Institut Teknologi Bandung. He holds a Japanese Patent on a high quality video compression system. His research interests include VLSI design, signal and image processing, VLC, smart cards, and electronics solution design and integration.



ADIWENA PUTRA received the B.Eng. degree in electrical engineering *(cum laude)* from Institut Teknologi Bandung, Indonesia, in 2020. He currently works as a Researcher at the University Center of Excellence on Microelectronics, ITB. His current research interests include VLSI design, computer architecture, hardware accelerator, and artificial intelligent.

NANA SUTISNA (Member, IEEE) received the B.Eng. degree in electrical engineering and the M.S. degree in microelectronics from the Bandung Institute of Technology, Indonesia, in 2005 and 2011, respectively, and the Ph.D. degree in computer science and electronics from the Kyushu Institute of Technology, in 2017. From 2017 to 2020, he was a Postdoctoral Fellow with the Department of Computer Science and System Engineering, Kyushu Institute of Technology. He is currently a Lecturer with Institut Teknologi Bandung. His research interests include VLSI design, baseband wireless system design, AI processor design, and HW/SW co-design and co-verification.

INFALL SYAFALNI (Member, IEEE) received the B.Eng. degree in electrical engineering from Institut Teknologi Bandung (ITB), Bandung, Indonesia, in 2008, the M.Sc. degree in electronic engineering from the University of Science Malaysia (USM), Penang, Malaysia, in 2011, and the Dr.Eng. degree in engineering from the Kyushu Institute of Technology (KIT), Iizuka, Fukuoka, Japan, in 2014. From 2014 to 2015, he has held a research position with KIT. From 2015 to 2018, he was an ASIC Engineer with the ASIC Development Group, Logic Research Company Ltd., Fukuoka. In 2019, he joined ITB, where he is currently an Assistant Professor with the School of Electrical Engineering and Informatics and a Researcher at the University Center of Excellence on Microelectronics. His current research interests include logic synthesis, logic design, VLSI design, and efficient circuits and algorithms.

RAHMAT MULYAWAN (Member, IEEE) received the B.Eng. degree in EE from ITB, Indonesia, in 2008, and the M.Sc. degree in EE from TU Delft, The Netherlands, in 2011. He is currently a member of the Microelectronics Center, ITB. His research interests include intelligent signal processing, MIMO systems, and transceiver design for optical wireless communications.