IEICE TRANSACTIONS

on Fundamentals of Electronics, Communications and Computer Sciences

DOI:10.1587/transfun.2024VLP0009

Publicized:2024/09/20

This advance publication article will be replaced by the finalized version after proofreading.

A PUBLICATION OF THE ENGINEERING SCIENCES SOCIETY

The Institute of Electronics, Information and Communication Engineers
Kikai-Shinko-Kaikan Bldg., 5-8, Shibakoen 3chome, Minato-ku, TOKYO, 105-0011 JAPAN

PAPER

# An FPGA-based YOLOv6 Accelerator for High-Throughput and Energy-Efficient Object Detection

**Xingan SHA**[†], *Nonmember*, **Masao YANAGISAWA**[†], *Fellow*, *and* **Youhua SHI**[†], *Member*

**SUMMARY** Fast, accurate, and energy-efficient object detection is increasingly important for edge applications, such as Internet of Things (IoT) devices. Among various convolutional neural network (CNN)-based methods, the You-Only-Look-Once (YOLO) algorithm series is regarded as one of the promising methods for real-time object detection due to its optimal balance between speed and accuracy. However, deploying YOLO on resource and power-constrained devices like field-programmable gate arrays (FPGAs) poses significant challenges due to the high demand for multiply-and-accumulate (MAC) operations and the corresponding significant off-chip memory accesses. This paper introduces an FPGA-based accelerator for the YOLOv6 algorithm, implemented on a VC707 FPGA board with a Virtex-7 VX485T chip, achieving satisfying throughput, accuracy, and energy efficiency. To our knowledge, this is the first FPGA implementation of YOLOv6. Unlike previous works that utilized early YOLO versions, our design deploys the hardware-friendly YOLOv6, achieving a mean average precision (mAP) of 84.9% on the PASCAL VOC2007 dataset at a 352*352 resolution - significantly outperforming most existing object detection implementations. Through model optimizations for FPGA deployment, such as changing from SiLU to ReLU activation, lowering input resolution, and applying quantization-aware training, we are able to greatly reduce computational cost with minimal accuracy loss. Furthermore, these optimizations allow for the entire YOLOv6 model to be stored in on-chip memory, eliminating the need for energy-intensive DRAM access. The proposed accelerator design and the convolution lowering technique also contribute to high processing speed and energy efficiency. Experimental results demonstrate that our accelerator can process 364.5 frames per second (fps) at 150 MHz on the Virtex-7 VX485T FPGA, achieving excellent power efficiency of 19.75 fps/W.

***key words:*** *Object Detection, CNN, YOLOv6, FPGA, Accelerator*

## 1. Introduction

Object detection is a technique in computer vision, used to locate and recognize objects within images or videos as shown in Fig. 1. Nowadays, CNN-based algorithms with exceptional accuracy are dominant in object detection with the help of Graphics Processing Units (GPUs) featuring powerful parallel computing capabilities. Traditional CNN-based algorithms like Region-based Convolution Neural Networks (R-CNNs) [1], and their variants, Fast R-CNN [2] and Faster R-CNN [3], are two-stage detection algorithms, which consist of generating potential bounding boxes, classification on these boxes and eliminating duplicate bounding boxes [4]. These algorithms are highly accurate but very time-consuming for the multiple stages, which are not suitable for real-time applications.

To address the trade-off between speed and accuracy in object detection, Redmon et al. introduced the one-stage detection algorithm, YOLOv1, in 2016 [4]. This innovative approach unified location and classification tasks into a single regression problem, leveraging a single neural network to generate bounding boxes and class probabilities simultaneously. As depicted in Fig. 2, the YOLO series has undergone significant evolution through versions [5], [6], [7], [8], [9], to [10], with the latest models, YOLOv5, YOLOv6, and YOLOv7, achieving accuracies of 55.8%, 52.5%, and 56.8% on the COCO dataset respectively. In this paper, accuracy specifically refers to mAP. These works represent a substantial improvement over the earlier versions like YOLOv3 and YOLOv4. Among these, YOLOv6 distinguishes itself by its hardware-friendly design. Furthermore, YOLOv6 is also quantization friendly, exhibiting minimal accuracy loss following Post Training Quantization (PTQ) or Quantization Aware Training (QAT). Notably, the YOLOv6-Nano version [9] stands out for its simplified structure, which significantly reduces memory requirements.

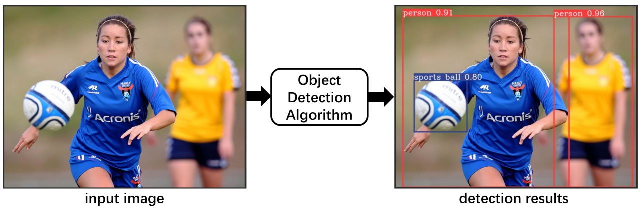


**Fig. 1** One example of object detection.

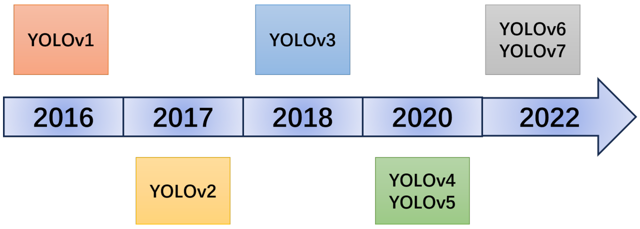


**Fig. 2** A timeline of YOLO series.

GPUs are widely used in deploying CNN algorithms like YOLO, but they are expensive and power consuming and not suitable for edge applications. Field-Programmable Gate Arrays (FPGAs) and Application-Specific Integrated Circuits (ASICs) are more appealing choices for lower cost and higher energy efficiency. FPGAs have advantages over ASICs in regarding of reconfigurability, and short development cycles. Therefore, this study aims to develop an FPGA-based YOLOv6 accelerator. FPGA-based DNN accelerators

[†]Waseda University, Shinjuku, Tokyo 169-8555, Japan

usually consist of off-chip memory (usually DRAM), on-chip global buffer (GLB), array of Processing Elements (PEs) composed of ALU and register file (RF), and inter-PE network [11]. The multiple PEs give designers opportunities of exploiting parallelism to improve performance. And DRAM, GLB, inter-PE network and RFs constitute a memory hierarchy. The DRAM access is very power-consuming and its energy cost is orders of magnitude higher than one GLB buffer access and one MAC computation. And the frequent DRAM access takes up a significant portion of energy cost in CNN accelerators. For example, the DianNao [12] and Cambricon-X [13] are state-of-the-art CNN accelerators, but their DRAM access energy costs account for 90% and 80% of the total energy costs, respectively. Therefore, the energy-consuming DRAM access is the bottleneck of deploying FPGA-based YOLO accelerators in power-constrained edge device.

There have been works for FPGA-based CNN accelerators, tailored for YOLO deployment. In [14], Yu et al. deployed YOLOv1 [4] on FPGA, and the layer fusion technique [15] is proposed to reduce the DRAM access. The YOLO accelerator mentioned in [14] managed to reach a performance of 15.4 frames per second (fps) while consuming 13W of power on the XILINX KU115, and it recorded a mAP of 62% on the VOC2007 dataset. In their study [16], Nguyen et al. effectively quantized YOLOv2 [5] to binary weights and 3-to-6 bit activations in a mere 2.63% accuracy decrease on the VOC2007 dataset. This approach allowed storing all model data in FPGA's BRAMs, cutting out DRAM accesses to save power. Their FPGA accelerator reached 109.3 fps and 18.29 W on the VC707 board with a 64.16% mAP on VOC2007. The authors in [17] developed an YOLOv3 accelerator on ZCU104 FPGA board and used the Vitis AI quantization tool to quantize FP32 network into int8 network, and its performance and power are 206.7 fps and 25W respectively. However, most of the existing works still rely on frequent DRAM access leading to not very good energy efficiency. In addition, existing works primarily utilize earlier YOLO versions, characterized by their significant computational demands and comparatively modest accuracy. The YOLOv6 [9] featuring high accuracy and hardware-friendly was introduced in 2022, yet, to date, there has been a lack of studies on FPGA-based accelerators for YOLOv6.

This work makes three contributions to the field. Firstly, this work deploys advanced YOLOv6 on FPGA for the first time to our knowledge, and the accuracy of this accelerator far exceeds most existing object detection implementations. Secondly, this work optimizes the YOLOv6 model for efficient FPGA implementations with little accuracy loss, including changing from SiLU to ReLU activation, lowering input resolution, and applying 6-bit integer quantization aware training. These model optimizations significantly cut down on hardware overhead and computational amount. Furthermore, these optimizations allow for the entire YOLOv6 model to be stored in on-chip memory, eliminating the need for power-hungry DRAM access for parameters and intermediate results. Thirdly, this work presents a novel accelerator design on FPGA, that leverages an output stationary dataflow and incorporates a true dual-port Psum Buffer and a pair of ping-pong Fmp Buffers, to achieve reduced latency and a compact memory footprint. To address the low PE array utilization observed in YOLOv6's initial convolutional layer, a convolution lowering technique is proposed, further lowering latency and diminishing the need for DRAM access.

The remainder of this paper is structured as follows: Section 2 provides an overview of the YOLOv6 algorithm and details the model optimizations we have applied for efficient hardware implementation. Section 3 describes the FPGA-based design of our YOLOv6 accelerator in details. Section 4 discusses the experimental results and compares our work with previous studies. Finally, conclusions are given in Section 5.

## 2. YOLOv6 Optimization for FPGA Implementation

In this section, the algorithm of YOLOv6 is firstly reviewed. And then the proposed model optimization techniques such as changing from SiLU to ReLU activation, lowering input resolution and applying quantization-aware training for efficient FPGA implementation are presented.

### 2.1 Review of YOLOv6 Algorithm

YOLOv6 was proposed in 2022 by Li et al. in [9]. YOLOv6 offers versions of different sizes, namely: YOLOv6-Nano, YOLOv6-Small, YOLOv6-Middle and YOLOv6-Large, to adapt for diversified scenarios. In the same GPU, YOLOv6 at different scales achieves higher accuracy, lower latency and higher throughput than other advanced YOLO object detectors like YOLOv5 [8] and YOLOv7 [10] at the same scale. In addition, YOLOv6 is friendly for quantization and has negligible accuracy loss after quantization, which makes it ready for deployment in industrial applications.

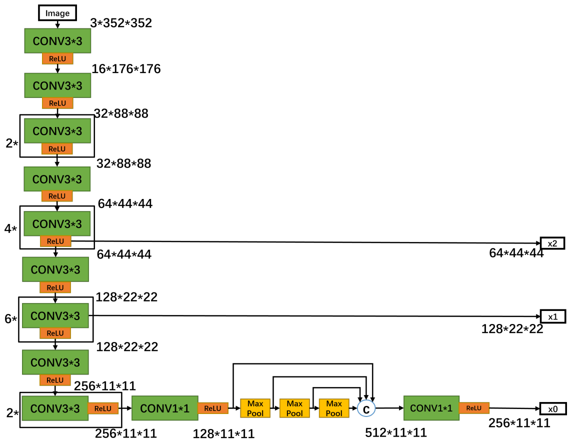


**Fig. 3** The structure of backbone of YOLOv6-Nano in inference.

YOLOv6 consists of three main components: a backbone, a neck, and a head. The backbone processes input

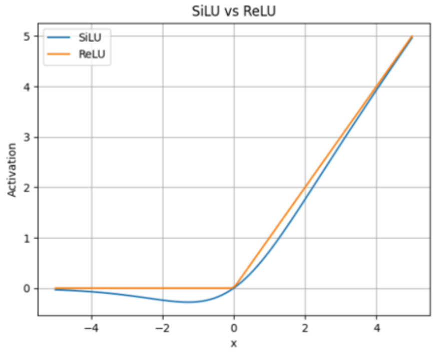


**Fig. 4** The activation functions of the SiLU and the ReLU.

images to extract features, accounting for the majority of the computation. The neck combines both low-level and high-level features to create pyramid feature maps. Finally, the head utilizes these pyramid feature maps to generate the final prediction results. As shown in Fig. 3, the YOLOv6-Nano backbone has a plain structure, which helps to reduce memory cost and is suitable for resource-constrained platforms like FPGA [9]. In addition, YOLOv6-Nano achieves higher throughput and lower latency than Small/Middle/Large and has the smallest model size [9]. Therefore, this work will deploy YOLOv6-Nano on FPGA for high speed object detection.

## 2.2 Changing SiLU to ReLU in Detection Head

In YOLOv6-Nano, all nonlinear functions in the backbone and neck are ReLU except the detection head using SiLU [18]. The output activation of the SiLU is obtained by multiplying the sigmoid function with its input activation [18], as shown in (1). Compared to SiLU, ReLU is a more simple computation as shown in Fig. 4.

$$\mathrm{SiLU}(x) = x \cdot \frac{1}{1+e^{-x}} \tag{1}$$

$$\mathrm{ReLU}(x) = \max(0, x) \tag{2}$$

If the input activation is positive or zero, the output activation of ReLU is equal to the input activation. If the input activation is negative, the output of ReLU is zero, as shown in (2). So, ReLU is easy to be realized in FPGA by only detecting whether the sign bit of input activation is 1 or 0. However, it is difficult to realize the accurate SiLU function in FPGA due to its complicated computation. SiLU's implementation on FPGA can consume an order of magnitude more logic resources than ReLU, with higher latency and high power consumption.

In our experiments for images with various resolutions as shown in Fig. 5, we have demonstrated that changing SiLU to ReLU only introduces a slight accuracy loss in VOC2007. Therefore, this work will replace all SiLUs in the head of YOLOv6-Nano with ReLUs as shown in Fig. 6.

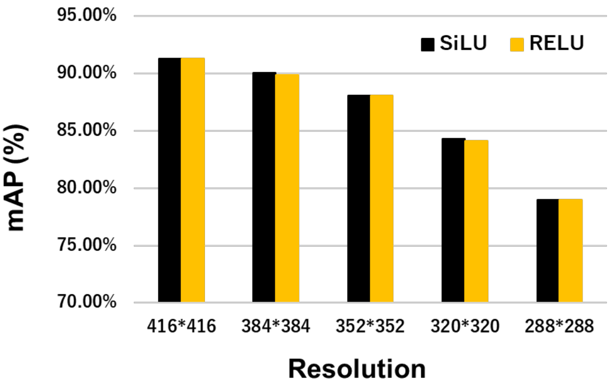


**Fig. 5** The mAPs of YOLOv6-Nano in various resolutions and activation functions in VOC2007.

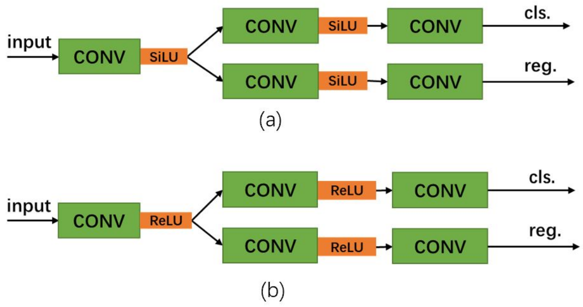


**Fig. 6** (a)The original head of YOLOv6[9] (b)The modified ReLU-head in this work.

## 2.3 Exploring Lower Resolution

Most of images' size in VOC2007 is 300*300 or less than 300*300. In previous studies, the image size is duplicated to 416*416 to ensure higher accuracy. However, this increase in size not only expands memory footprint but also escalates computational amount, which has a negative impact on latency. As illustrated by the blue curve in Fig. 7, the accuracy changes relatively slowly before reaching 352*352, but after that, it drops dramatically. The yellow curve in the same figure highlights that lowering the resolution can effectively reduce computational amount, thus improving processing speed.

In this experiment as shown in Fig. 7, we have demonstrated that lowering resolution significantly reduce latency but there is a risk of a sudden drop in accuracy. Therefore, this work sets image size to 352*352 for the optimal balance between accuracy and speed.

## 2.4 Quantization

Quantization is the process of transforming values from a continuous or wide range into a discrete and narrower range [11], essentially reducing bits used to represent data. This reduction in data bitwidth can effectively reduce hardware

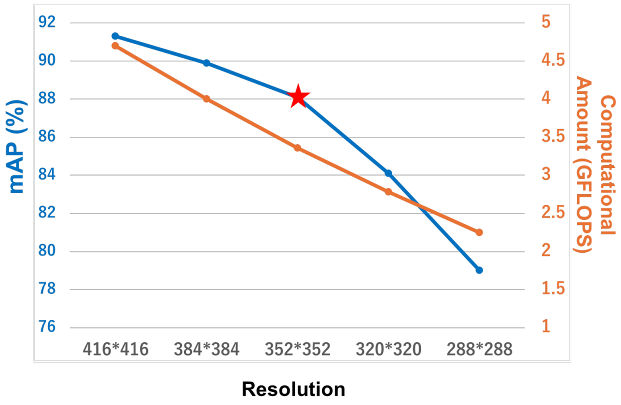


**Fig. 7** The mAPs and computational amount of YOLOv6-Nano in different resolution.

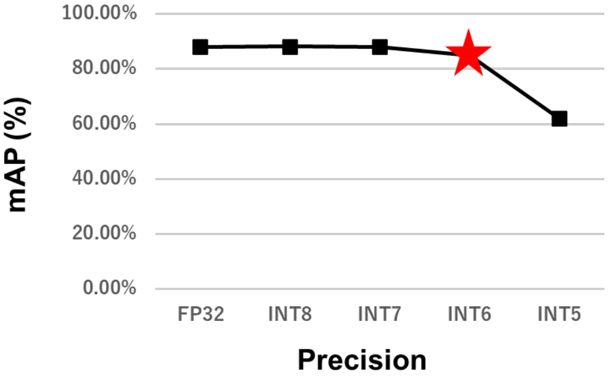


**Fig. 8** The mAPs of YOLOv6-Nano in different quantization schemes.

overheads, memory footprint and energy costs, which is suitable for resource-power-constrained platforms like FPGA. While most mainstream quantization tools standardize on 8-bit integers, FPGAs offer the flexibility to customize bitwidth, enabling the model in precision of less than 8-bit integer. To enhance resource efficiency and energy savings, this work would try to quantize the model in the precision of less than 8 bits. The Pytorch-Quantization package developed by Nvidia is employed to facilitate quantization-aware training in this work. This method is a form of linear quantization, where both the weights and activations are represented as signed integers.

In our evaluation of different quantization schemes or precisions shown in Fig. 8, the int5 quantization scheme led to a significant drop in accuracy, deeming it impractical. While the int8 and int7 quantization schemes maintained high accuracy, the models in the int8 and int7 precisions cannot be accommodated within the FPGA's on-chip memory and thus heavily rely on energy-intensive DRAM access. On the other hand, the model in the int6 quantization scheme can be entirely stored within on-chip memory, eliminating all DRAM access of parameters and intermediate results. Therefore, this work adopts int6 quantization scheme for the optimal balance between accuracy and energy efficiency.

## 3. Proposed Accelerator Design

This section outlines the architecture of the FPGA-based accelerator, highlighting components like the true dual-port Psum Buffer and ping-pong Fmp Buffers used to store intermediate data. It also covers the configuration of the PE (Processing Element) array and the corresponding output-staionary dataflow. Lastly, we introduce a convolution lowering technique aimed at addressing the low PE array utilization observed in YOLOv6's initial convolutional layer, which contributes to reduced latency and fewer DRAM accesses.

### 3.1 The Architecture of the Accelerator

Fig. 9 depicts the architecture of the FPGA-based YOLOv6-Nano accelerator, which includes Processor coordinating and controlling all modules, a Conv Module performing the convolution and max pooling computations, and three ROMs for weights, biases, and quantization coefficients (q_cos). Fmp Buffers 0/1 store the feature map (fmp), which is the output of a convolutional layer, while the Psum Buffer holds partial sums (psums), the intermediate results that are computed prior to the generation of a fmp. The Direct Memory Access (DMA) unit is a circuit used to enhance data transfer efficiency between DRAM and Fmp Buffers, receiving control signals from the Processor. In this design, the DMA unit is responsible for reading image from DRAM to Fmp Buffer and writing the results from Fmp Buffer to DRAM. This accelerator features eliminating all DRAM accesses of psums and fmps. And this is attributed to the abundant on-chip memory of FPGA and the above-mentioned int6 quantization scheme, so that more BRAMs are conserved for the Fmp Buffers 0/1 and Psum Buffer. And they are large enough to accommodate the largest fmp and psum that aid in bypassing DRAM accesses for intermediate data. The Fmp Buffer 0 and 1 are also ping-pong buffers, which helps to reduce latency. The roles of Fmp Buffer 0 and 1 are swapped in the next convolutional layer, so the output of one layer can be directly used as input in the next layer. The Psum Buffer is one true dual port RAM, which can be read and written at the same time. In processing one convolutional layer, the computation is usually divided into several tiles along the channel dimensions. And the Conv Module computes the convolution tile by tile. The generated psums of the last tile will be read from the Psum Buffer as the input psums (ipsums) for the Conv Module, and the psums computed in this tile are simultaneously written into the Psum Buffer as output psums (opsums). Within the Psum Buffer, since the read and write operations on the same data location are staggered in time, the convolution computation results remain accurate even without having two separate input and output Psum Buffers. A single true dual-port buffer effectively achieves the ping-pong effect of two buffers, which helps to reduce latency and save memory at the same time. Therefore, this architecture has advantages over other works in throughput and energy efficiency, as it eliminates almost

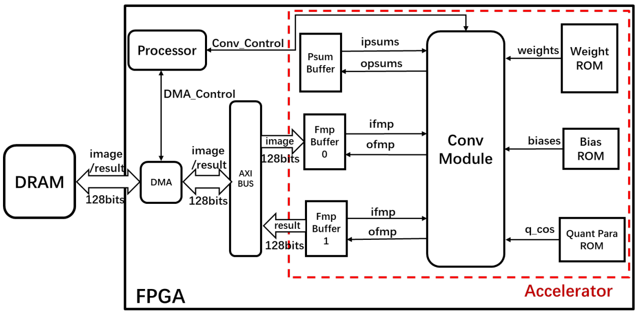


**Fig. 9** The architecture of this accelerator.

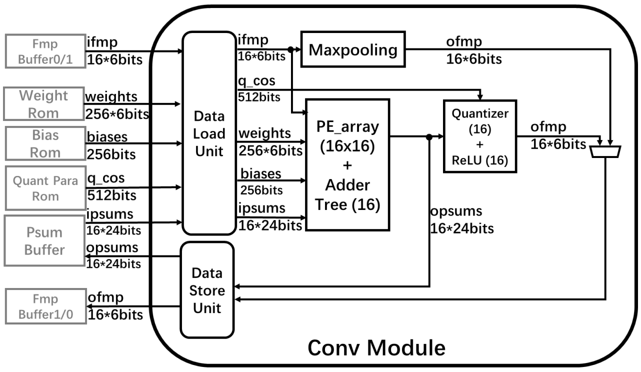


**Fig. 10** The structure of the Conv Module.

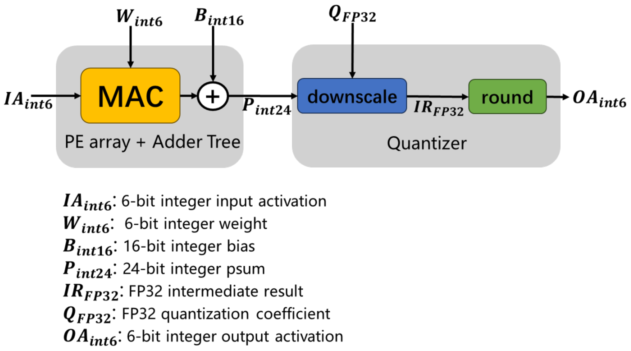

$W_{int6}$
$B_{int16}$
$Q_{FP32}$
$IA_{int6}$
MAC
+
$P_{int24}$
downscale
$IR_{FP32}$
round
$OA_{int6}$
PE array + Adder Tree
Quantizer
$IA_{int6}$: 6-bit integer input activation
$W_{int6}$: 6-bit integer weight
$B_{int16}$: 16-bit integer bias
$P_{int24}$: 24-bit integer psum
$IR_{FP32}$: FP32 intermediate result
$Q_{FP32}$: FP32 quantization coefficient
$OA_{int6}$: 6-bit integer output activation

**Fig. 11** The data processing of the integer-based convolution in this work, and it is handled by the PE array and quantizer within the Conv Module.

all DRAM accesses and adopts ping-pong buffers.

The Processor sends control signals to the Conv Module, including the start signal, convolutional information, and address information. Convolutional information comprises kernel size, stride, fmp size, and controls for reading ipsums from and writing opsums to the Psum Buffer, enabling the Conv Module to perform multiple convolutions flexibly and accurately. The address information includes the addresses for fmps and parameters. Specifically, the fmp address information covers the ifmp and ofmp addresses, along with the Fmp Buffer state, which indicates whether a buffer is an input or output. This setup ensures the Conv Module correctly reads the ifmp from the input Fmp Buffer and writes the ofmp to the appropriate location in the output Fmp Buffer. The parameter information furnishes the Conv Module with addresses in the Weight ROM, Bias ROM, and Quantization Parameter ROM for the weights, biases, and q_cos necessary for the convolutional layer it processes. During a convolution, the Processor provides the Conv Module with both convolutional and address information. Upon receiving this information, the Processor raises the 1-bit start signal from 0 to 1. The Conv Module, detecting the rising edge of the Start signal, initiates the convolution or max pooling operation. It reads the ifmp and ipsums from Fmp Buffer 0 (1) and the Psum Buffer, respectively, and retrieves corresponding weights, biases, and q_cos from the Weight ROM, Bias ROM, and Quantization Parameter ROM to perform the convolution. If all MAC operations of one convolutional layer are finished, the ofmp produced by the Conv Module are stored in the Fmp Buffer 1 (0). If not, the generated opsums are stored in the Psum Buffer. Once the computation is complete, the Conv Module sends a stop signal to the Processor, which then prepares control signals for the next convolution.

### 3.2 The Design of Conv Module

Fig.10 depicts the Conv Module's architecture, featuring several key components: the Data Load Unit, Maxpooling Unit, a 16*16 PE array with 16 18-input 4-pipeline-stage Adder Trees, 16 Quantizer plus ReLU Units, and the Data Store Unit. The Data Load Unit retrieves 16 channels of ifmp from Fmp Buffer 0/1, dispatching them to the PE array for convolution, along with 256 weights (16 input channels and 16 output channels), 16 biases, and 16 q_cos from its on-chip ROMs. The module also incorporates a Maxpooling Unit for max pooling operations, triggered by the corresponding command of the Processor. The 16*16 PE array and 16 Adder Trees execute the integer-based MAC and bias-addition operations of convolution as shown in Fig.11, producing 16 channels of 24-bit integer opsums simultaneously. The bit width of opsums is set to 24-bit to prevent overflow during the maximal accumulation of 6-bit by 6-bit multiplications of 256 input channels with a 3*3 kernel size. Once all the integer-based operations for a layer are complete, these 24-bit integer psums are mapped to 6-bit integer output activations of ofmp. This mapping process consists of 2 subprocesses (downscale and round) is executed by the Quantizer Unit as shown in Fig.11. First, the 24-bit opsum would be multiplied with one FP32 quantization coefficient to be downscaled and clipped to an FP32 intermediate result within the range of [-32, 31]. The q_cos are determined by the quantization tool of Nvidia to avoid large mAP decay. Then, the round process approximates the downscaled FP32 intermediate result to the nearest integer value, which is the output activation of ofmp. Lastly, the Data Store Unit archives the ofmp from the Quantizer or Maxpooling Unit into output Fmp Buffer. If the MAC operations of one convolutional layer are not finished, the Data Store Unit stores the PE array's intermediate opsums into the Psum Buffer.

### 3.3 The Structure of PE array

The network's convolutional layers feature variable kernel sizes of 1*1, 2*2, and 3*3, necessitating the sequential execution of loops of kernel size ($H_k$ and $W_k$). In order to minimize the storage of psums, this accelerator adopts output stationary dataflow, prioritizing the $H_o$ and $W_o$ loops over the $H_k$ and $W_k$ loops to accumulate most of psums within PEs. To enhance parallelism, the accelerator unrolls the loops for both input channel ($C_i$) and output channel ($C_o$) as depicted in Fig. 12 (a). Given the limited on-chip buffer, the $C_i$ and $C_o$ loops cannot be fully unrolled. Therefore, the $C_i$ and $C_o$ loops are divided into 4 loops, with tile sizes ($TC_i$ and $TC_o$). Notably, $TC_i$ corresponds to the width of the PE array, while $TC_o$ corresponds to the height of the PE array as shown in Fig. 12 (a). Fig. 12 (b) outlines the PE array structure, comprising 16*16 PEs and 16 18-input Adder Trees. Each PE includes one multiplier and one self-accumulator performing $H_k$*$W_k$ MACs within one input channel, while the 18-input Adder Tree integrates one ipsum, one bias, and the 16 outputs from 16 PEs in one row to get activations of one output channel in ofmp. The self-accumulator is set to 16 bits to prevent overflow, as there are up to nine 6-bit by 6-bit multiplications accumulated within a single PE. The PE array has 16 rows allowing it to compute 16 output channels of ofmp in parallel. This work chooses 16*16 PE array for the following reason. The dimensions of the PE array should be symmetrical due to the unrolling of both input and output channels, along with the employment of ping-pong buffers. For the YOLOv6-Nano model, the channel numbers of most convolutional layers are multiples of 16. In processing a convolutional layer with both input and output channels set at 32, a 32*32 PE array would theoretically be four times faster than a 16*16 array. However, a 31*31 PE array may not achieve a faster performance than a 16*16 array due to underutilization of PEs. To prevent this inefficiency, the PE array size should conform to multiples of 16, like 16*16 or 32*32. In this study, a 16*16 PE array was chosen as it nearly utilizes all available BRAM. Expanding to a 32*32 PE array would demand double the on-chip buffer. Thus, the 16*16 PE array provides a practical balance between performance and resource constraints.

### 3.4 Convolution Lowering for PE Utilization Improvement

The PE array is designed to simultaneously process input activations from 16 channels in a single clock cycle, yet an image typically contains only 3 channels. In addition, DRAM contents are often random after powering up, necessitating correct initialization for the first convolutional layer's accurate processing. To address this, the original 3*352*352 image is expanded on the host computer with 13*352*352 zero-value tensor, then transferred to the FPGA board's DRAM. This approach, however, results in a mere 18.75% efficiency in PE array usage, as depicted in Fig. 13. Moreover, the expanded 16*352*352 tensor requires 5.33 times more DRAM accesses than the original image.

To address the issue of small number of image channels, this study introduces a convolution lowering strategy, illustrated in Fig. 14, aimed at augmenting the number of image channels or input channels. During convolution lowering, the original $C_i$ k*k weight matrix in one output channel is split into $C_i$*k*k individual weights. Simultaneously, the ifmp's data is duplicated and rearranged following a specific pattern leading to the correct result of the original k*k convolution through a new 1*1 convolution. This approach effectively increases the input channels by a factor of k*k by transforming the original $C_i$-input-channel, k*k convolution into a new $C_i$*k*k-input-channel, 1*1 convolution, and ensures the ofmp remains consistent. Correspondingly, the initial $C_i*H_i*W_i$ ifmp is restructured into a new $(C_i*k*k)*H_o*W_o$ ifmp without affecting accuracy.

Applying the convolution lowering technique in YOLOv6-Nano transforms a 3*352*352 image into a 27*176*176 tensor. This tensor is then augmented with five channels of zero-value black images to create a 32*176*176 tensor. As a result, PE utilization surges to 84.375%, as depicted in Fig. 15. Despite the 32*176*176 tensor being 2.67 times larger than the original 3*352*352 image, it is still half the size of the 16*352*352 tensor produced without using the convolution lowering technique. Thus, this method effectively halves the DRAM accesses required to transfer the image to the FPGA board. Experimentally, processing the first convolutional layer without this technique takes 154,382 clock cycles, but only 78,557 with it. In terms of overall performance, the latency for executing all layers has been reduced by approximately 15.56%, and the throughput has increased by about 18.43%. Hence, convolution lowering not only halves DRAM accesses but also significantly reduces latency for the first convolutional layer.

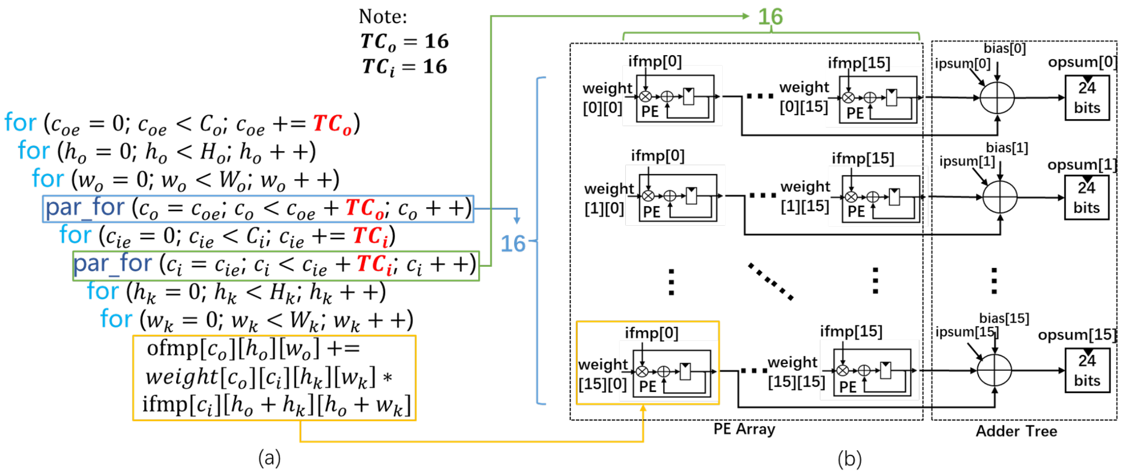
## 4. Experimental Results

### 4.1 Implementation Result

Our design has been deployed on a Xilinx VC707 board equipped with an XC7VX485T FPGA chip, as illustrated in Fig. 16 (a). The accelerator's architecture, detailed in Fig. 16 (b), operates at a frequency of 150 MHz, with the VC707 board consuming 18.46W of power at a supply voltage of 12V. Resource usage on the FPGA, shown in Fig. 17, reveals that over 94% of the Block RAMs (BRAMs) serve as on-chip buffers and ROMs, significantly reducing DRAM access needs. The composition of these on-chip storages is depicted in Fig. 18, with more than 70% dedicated to Parameter ROM for storing the entire model. Processing a single image incurs a total latency of 411,438 clock cycles, achieving a throughput of 364.5 frames per second (fps).

### 4.2 Comparison Result

Based on the FPGA implementation results, we compared



```
Note:
$TC_o = 16$
$TC_i = 16$

for ($c_{oe} = 0$; $c_{oe} < C_o$; $c_{oe}$ += $TC_o$)
  for ($h_o = 0$; $h_o < H_o$; $h_o++$)
    for ($w_o = 0$; $w_o < W_o$; $w_o++$)
      par_for ($c_o = c_{oe}$; $c_o < c_{oe} + TC_o$; $c_o++$)
        for ($c_{ie} = 0$; $c_{ie} < C_i$; $c_{ie}$ += $TC_i$)
          par_for ($c_i = c_{ie}$; $c_i < c_{ie} + TC_i$; $c_i++$)
            for ($h_k = 0$; $h_k < H_k$; $h_k++$)
              for ($w_k = 0$; $w_k < W_k$; $w_k++$)
                $ofmp[c_o][h_o][w_o]$ += $weight[c_o][c_i][h_k][w_k] * ifmp[c_i][h_o+h_k][h_o+w_k]$
```

**Fig. 12** (a) Loop reorder, tiling and parallel of convolution algorithm. (b) The structure of PE array and Adder Tree. (Note: The self-accumulator register within one PE is 16 bits.)

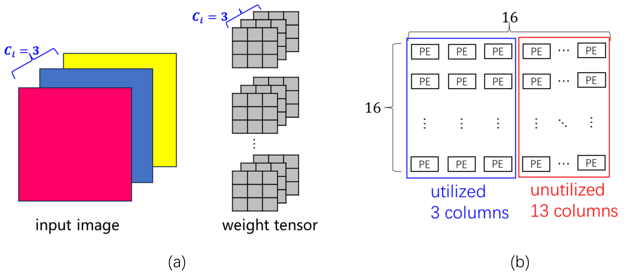


**Fig. 13** (a) The input image and weight tensor of the first convolutional layer. (b) The utilization of PE array is 18.75% when processing the first convolutional layer.

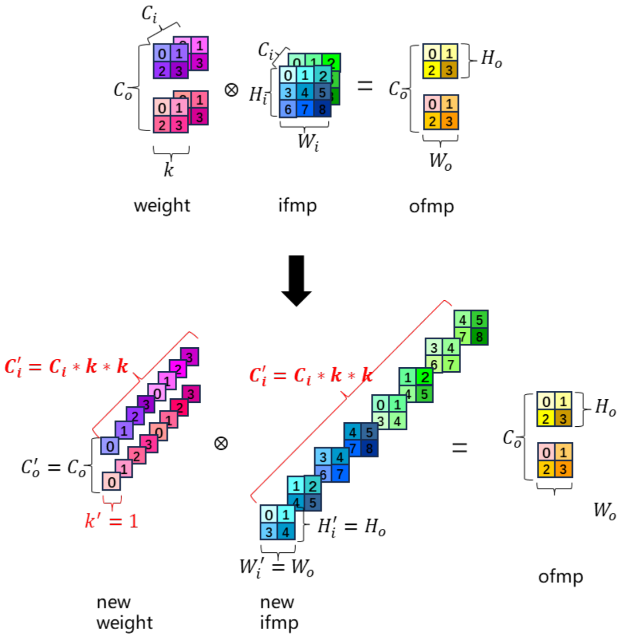


**Fig. 14** Convolution lowering: mapping a k*k convolution to a new 1*1 convolution to increase input channel number by decomposing weight matrix in one output channel and rearranging input feature maps.

our design with the existing accelerators in terms of detection accuracy, throughput and power efficiency. As shown in Table 1, this work achieves the highest accuracy in VOC2007 dataset, which is attributed to the advanced YOLOv6 algorithm. This design also achieves the highest throughput and power efficiency. There are three design advantages that contribute to achieving such excellent results. Firstly, the image resolution of this design is only 352*352, which significantly reduces energy-consuming DRAM accesses of input images and computational workloads. Secondly, all DRAM accesses of parameters and intermediate feature maps are eliminated, significantly increasing speed and reducing energy cost. Finally, most existing works adopt High Level Synthesis (HLS) tool or Vitis AI to improve development speed on FPGA. However, HLS tools and Vitis AI provide limited optimization opportunities on performance and resource utilization, leading to lower speed and higher resource cost. Therefore, this design is developed by the Verilog Hardware Design Language (HDL), which offers more accurate control over resource utilization and clock cycles.

## 5. Conclusions

This paper introduces a fast, energy-efficient and accurate object detection system implemented on Xilinx VC707, in which the state-of-the-art YOLOv6-Nano algorithm is deployed on FPGA through HDL for the first time. Leveraging model optimization techniques such as transitioning from SiLU to ReLU activations, reducing input resolution, and adopting quantization-aware training, we significantly decrease the model size. This allows for the entire model to be stored on the FPGA board, obviating the need for DRAM accesses for both model parameters and intermediate results. In addition, the proposed accelerator design and the convolution lowering technique also contribute to reduced latency and fewer DRAM access. Consequently, our system design

**Table 1** Comparison with other works.

| Design | Model | Image Size | Precision | mAP on VOC2007 | Model Size | Platform | Throughput (fps) | Power (W) | Power Efficiency (fps/W) |
|---|---|---|---|---|---|---|---|---|---|
| [14] | YOLOv1 | 416*416 | 8-bit fixed | 62% | - | Xilinx KU115 | 15.4 | 13 | 1.18 |
| [16] | Sim-YOLOv2 | 416*416 | w:1bit<br>a:3-6 bits | 64.16% | 1.88MB | Xilinx VC707 | 109.3 | 18.29 | 5.98 |
| [19] | YOLOv2 | 416*416 | 8-bit | 74.45% | 10MB | Intel Arria-10 GX1150 | 72.5 | 26 | 2.79 |
| [17] | YOLOv3 | 416*416 | 8-bit integer | - | - | Xilinx ZCU104 | 206.7 | 25 | 8.27 |
| [20] | YOLOv3 | 448*448 | 8-bit integer | 75% | - | Xilinx Ultra96 V2 | 19.2 | 5.44 | 3.53 |
| This work | YOLOv6 | 352*352 | 6-bit integer | 84.9% | 3.1MB | Xilinx VC707 | 364.5 | 18.46 | 19.75 |

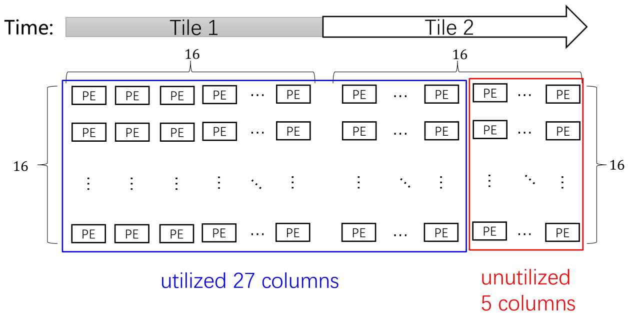


**Fig. 15** The utilization of PE array in processing the first convolutional layer is increased to 84.375% when adopting convolution lowering technique.

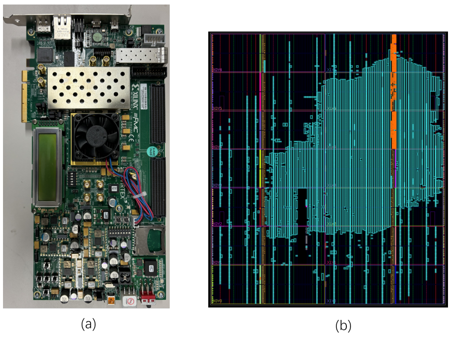


**Fig. 16** (a)The VC707 FPGA board with 485760 logic cells, 2800 DSPs and 37080 Kb memory, and (b)the layout of our accelerator design on VC707.

demonstrates superior performance over previous works in both speed and energy efficiency, while maintaining the highest accuracy in object detection. These results underscore the design's potential for application in environments where power and resources are limited, such as edge computing devices.

## References

[1] R. Girshick, J. Donahue, T. Darrell, and J. Malik, "Rich feature hierarchies for accurate object detection and semantic segmentation," in *Proc. IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2014, pp. 580–587.

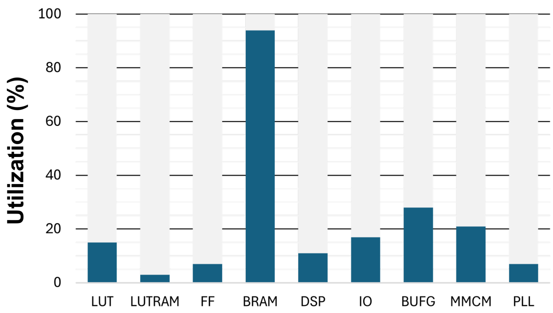


**Fig. 17** Resource utilization on VC707.

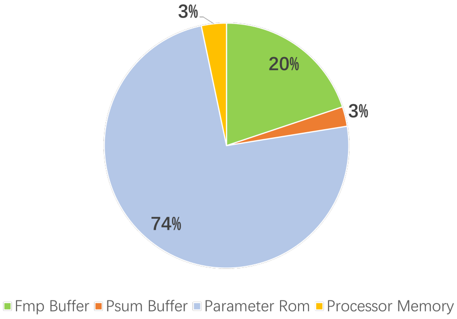


**Fig. 18** Breakdown of the on-chip memory.

[2] R. Girshick, "Fast R-CNN," in *Proc. IEEE International Conference on Computer Vision (ICCV)*, 2015, pp. 1440–1448.

[3] S. Ren, K. He, R. Girshick, and J. Sun, "Faster R-CNN: Towards real-time object detection with region proposal networks," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 39, no. 6, pp. 1137–1149, 2017.

[4] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi, "You only look once: Unified, real-time object detection," in *Proc. IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016, pp. 779–788.

[5] J. Redmon and A. Farhadi, "YOLO9000: Better, faster, stronger," in *Proc. IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2017, pp. 6517–6525.

[6] J. Redmon and A. Farhadi, "YOLOv3: An incremental improvement," *arXiv preprint arXiv:1804.02767*, 2018.

[7] A. Bochkovskiy, C.-Y. Wang, and H.-Y. M. Liao, "YOLOv4: Op-

timal speed and accuracy of object detection," *arXiv preprint arXiv:2004.10934*, 2020.

[8] G. Jocher, "YOLOv5 by ultralytics," https://github.com/ultralytics/yolov5, 2020.

[9] C. Li, L. Li, H. Jiang, K. Weng, Y. Geng, L. Li, Z. Ke, Q. Li, M. Cheng, W. Nie, Y. Li, B. Zhang, Y. Liang, L. Zhou, X. Xu, X. Chu, X. Wei, and X. Wei, "YOLOv6: A single-stage object detection framework for industrial applications," *arXiv preprint arXiv:2209.02976*, 2022.

[10] C.-Y. Wang, A. Bochkovskiy, and H.-Y. M. Liao, "YOLOv7: Trainable bag-of-freebies sets new state-of-the-art for real-time object detectors," *arXiv preprint arXiv:2207.02696*, 2022.

[11] M. Capra, B. Bussolino, A. Marchisio, G. Masera, M. Martina, and M. Shafique, "Hardware and software optimizations for accelerating deep neural networks: Survey of current trends, challenges, and the road ahead," *IEEE Access*, vol. 8, pp. 225 134–225 180, 2020.

[12] T. Chen, Z. Du, N. Sun, J. Wang, C. Wu, Y. Chen, and O. Temam, "DianNao: A small-footprint high-throughput accelerator for ubiquitous machine-learning," *ACM SIGARCH Computer Architecture News*, vol. 42, no. 1, pp. 269–284, 2014.

[13] S. Zhang, Z. Du, L. Zhang, H. Lan, S. Liu, L. Li, Q. Guo, T. Chen, and Y. Chen, "Cambricon-X: An accelerator for sparse neural networks," in *Proc. Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2016, pp. 1–12.

[14] J. Yu, K. Guo, Y. Hu, X. Ning, J. Qiu, H. Mao, S. Yao, T. Tang, B. Li, Y. Wang, and H. Yang, "Real-time object detection towards high power efficiency," in *Proc. Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2018, pp. 704–708.

[15] M. Alwani, H. Chen, M. Ferdman, and P. Milder, "Fused-layer CNN accelerators," in *Proc. Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2016, 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 Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 27, no. 8, pp. 1861–1873, 2019.

[17] J. Wang and S. Gu, "FPGA implementation of object detection accelerator based on Vitis-AI," in *Proc. International Conference on Information Science and Technology (ICIST)*, 2021, pp. 571–577.

[18] S. Elfwing, E. Uchibe, and K. Doya, "Sigmoid-weighted linear units for neural network function approximation in reinforcement learning," *Neural networks*, vol. 107, pp. 3–11, 2018.

[19] Z. Wang, K. Xu, S. Wu, L. Liu, L. Liu, and D. Wang, "Sparse-YOLO: Hardware/software co-design of an FPGA accelerator for YOLOv2," *IEEE Access*, vol. 8, pp. 116 569–116 585, 2020.

[20] T. Adiono, A. Putra, N. Sutisna, I. Syafalni, and R. Mulyawan, "Low latency YOLOv3-tiny accelerator for low-cost FPGA using general matrix multiplication principle," *IEEE Access*, vol. 9, pp. 141 890–141 913, 2021.

**Xingan Sha** received the M.Eng. degree from Waseda University in 2023 and presently works toward Dr. Eng. degree there. His research interests include energy-efficient hardware acceleration of neural networks.

**Masao Yanagisawa** received the B. Eng., M. Eng. and Dr. Eng. degrees from Waseda University in 1981, 1983, and 1986, respectively, all in electrical engineering. He was with University of California, Berkeley from 1986 through 1987. In 1987, he joined Takushoku University. In 1991, he left Takushoku University and joined Waseda University, where he is presently a Professor in the Faculty of Science and Engineering. His research interests are combinatorics and graph theory, computational geometry, LSI design and verification, and bioinformatics. He is a member of IEEE, ACM, IEICE, IPSJ, and Operations Research Society of Japan.

**Youhua Shi** received the Dr.Eng. degree in Electronics, Information and Communication Engineering from Waseda University in 2005. He is currently a professor of Waseda University. His current research interests include energy harvesting, intelligent computing and trustworthy computing. He is a member of IEEE, IEICE and IPSJ.