IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Received May 1, 2020, accepted June 14, 2020, date of publication June 22, 2020, date of current version July 3, 2020.

Digital Object Identifier 10.1109/ACCESS.2020.3004198

Sparse-YOLO: Hardware/Software Co-Design of an FPGA Accelerator for YOLOv2

ZIXIAO WANG[1,2], KE XU[1,2], SHUAIXIAO WU[1,2], LI LIU[3], LINGZHI LIU[3], (Senior Member, IEEE), AND DONG WANG[1,2], (Member, IEEE)

[1]Institute of Information Science, Beijing Jiaotong University, Beijing 100044, China
[2]Beijing Key Laboratory of Advanced Information Science and Network Technology, Beijing 100044, China
[3]Heterogeneous Computing Group, Kuaishou Technology, Palo Alto, CA 94306, USA

Corresponding author: Dong Wang (wangdong@bjtu.edu.cn)

This work was supported in part by the National Key Research and Development Program of China under Grant 2019YFB2204200, in part by the Beijing Natural Science Foundation under Grant 4202063, and in part by BJTU-Kuaishou Research Grant.

ABSTRACT Convolutional neural network (CNN) based object detection algorithms are becoming dominant in many application fields due to their superior accuracy advantage over traditional schemes. Among them, You Look Only Once (YOLO) is one of the most popular detection frameworks that show best trade-offs between speed and accuracy. However, due to the intrinsic high computational workload of CNN, it is still challenging when targeting high-throughput processing with low cost in energy consumption. In this paper, we propose a hardware/software (HW/SW) co-design methodology targeting CPU+FPGA-based heterogeneous platforms. Firstly, we extend a novel sparse convolution algorithm to the YOLOv2 framework, and then develop a resource-efficient FPGA accelerator architecture based on asynchronously executed parallel convolution cores. Secondly, algorithm-level optimization schemes, including hardware-aware neural network pruning, clustering and quantization are introduced, which successfully save the computational workload of the YOLOv2 algorithm by 7 times. Finally, an end-to-end design space exploration flow for FPGA-based accelerator design is presented and two HW/SW partition strategies are studied and implemented. Experimental results show that our design can achieve a peak throughput of 2.13 TOPS (72.5 fps) on an Intel Arria-10 GX1150 FPGA under the working frequency of 211 MHz, while the detection accuracy is 74.45 on the PASCAL VOC2007 dataset.

INDEX TERMS Convolutional neural networks, fine-grained pruning, field programmable gate arrays, object detection, YOLO.

I. INTRODUCTION

Convolutional neural networks (CNNs) based object detection approaches [1]–[5] have shown remarkable performance advantages over traditional methods [6]–[8]. Although CNNs are compute-intensive, recent researches have shown that the computational workload and memory bandwidth requirements can be significantly reduced by performing weight pruning [9]–[12] and quantization [13]–[15] on the CNN model used for inference computation. At hardware level, dedicated accelerators/processors [16]–[21] have also been studied to exploit the inherent parallelism of the convolution algorithm to further speed-up the computation. Among all the hardware platforms, field-programmable gate arrays (FPGAs) have received increasing attention due to its flexible architecture (including massive processing elements, on-chip memory blocks and reconfigurable interconnections), very low power consumption and fast development cycle time (especially with the help of high-level-synthesis (HLS)-based tools).

There have been several FPGA-based accelerator designs that were specially developed to implement the You Look Only Once (YOLO) series algorithms in the literature [17], [18], [22]–[25]. According to the type of convolution algorithm implemented, we can divide these designs into three broad categories: The first type of design exploits the inherent parallelism of the CNN inference computation in a straightforward way of performing spatial convolution algorithms

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

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

with a massive number of multiply-accumulate (MAC) operations on large numbers of Digital Signal Processor (DSP) blocks. For instance, the accelerator presented in [22] implemented the YOLOv2 network on an Intel Arria-10 FPGA achieving a throughput of 566 GOPS (i.e., 18.86 fps at 416×416 input resolution). Another design [26] accelerated the Tiny-YOLO network on a Xilinx Virtex7-485t FPGA at the detection speed of 21 fps. It has been shown by previous studies [27] that one important limitation of the spatial-convolution-based FPGA accelerators is that the highest attainable performance is solely determined by the processor's hardware structure, i.e., the number of DSP blocks (MAC units) used.

Another category of schemes replaces the DSP demanding MAC operations required by the convolution computation with low-bit or single-bit logic operations, such as AND or XOR, so that FPGA's logic resource, which is generally more abundant than the DSP blocks, can be utilized to develop more efficient hardware circuit that can break the limitation of the on-chip DSP resource. For instance, in [18], the authors optimized a binarized CNN backbone network for feature extraction in YOLOv2. They implemented a resource-efficient design on a Xilinx Zynq Ultrascale FPGA, achieving a high processing speed of 40.81 fps. Similarly in [23], an FPGA accelerator that implemented a mixed precision YOLOv2 network model was developed and achieved comparable detection performance to that of [18]. However, one significant drawback of using low precision quantized neural network is that serious loss of detection accuracy has to be tolerated.

The third type of accelerators utilize advanced convolution algorithms, such as frequency domain convolution schemes [17], [24], [25], in hardware design to reduce the computational workload (i.e., the number of MAC operations) compared to the straightforward spatial convolution algorithm, and, in return, obtains considerable performance gain for the implemented accelerator. In [24], the accelerator design utilized the Winograd [28] convolution algorithm and gained a 47% workload reduction for the YOLOv1 algorithm at the cost of only 1% degradation in detection accuracy. The final implementation achieved a 15.3 fps detection speed (969 GOPS throughput) and consumed 1024 DSP blocks on a Xilinx KU115 FPGA. The authors in [25] extended the Winograd approach to accelerate the YOLOv2 network. However, the presented design adopted 16-bit precision for CNN model and 32-bit precision for convolution data-path, which doubled the hardware cost comparing to the 8-bit quantization-based design of [22]. Therefore, the reported performance of [25] is very similar to that of [22] on the same FPGA board. The accelerator design of [17] adopted an FFT-based circulate convolution algorithm for the tiny-YOLO network and achieved a $15\times$ higher throughput over previous FPGA implementation [26].

Based on above analysis, we can conclude that employing software-level optimization of the convolution algorithm to trim the workload of the CNN model without introducing obvious degradation on the object detection accuracy is a more promising way than the other two approaches. Moreover, beside using frequency-domain convolution algorithm, there is also another type of advanced convolution scheme named sparse convolution, which has been reported by recent studies of [27], [29], [30], showing even better performance over frequency-domain approach in designing image-classification-targeted CNN inference accelerator on FPGA.

However, such scheme has not yet been applied to the YOLO series object detection algorithms, which have more complicated processing flows than classification-based CNNs. Therefore, in this paper, we extend the sparse convolution algorithm introduced by our previous study of [27] to the YOLOv2 algorithm and develop a resource-efficient hardware architecture on FPGA targeting at improved computation throughput over state-of-the-art YOLO accelerators. The key contributions of this study include:

- A novel sparse convolution scheme, namely the Accumulate-Before-Multiplication Sparse-Convolution (ABM-SpConv), was applied to the YOLOv2 object detection algorithm. We proposed a hardware-aware algorithm-level optimization flow for YOLOv2 network including pruning, clustering, layer fusion, and quantization to overcome the low hardware utilization issue of the ABM-SpConv scheme that the study of [27] has failed to address.
- A dedicated accelerator architecture was developed targeting for CPU+FPGA-based heterogeneous computing platforms. The proposed accelerator architecture achieved the best balance between the flexibility of software and the high computational efficiency of customized hardware circuits.
- An end-to-end hardware-software (HW-SW) co-design work flow was proposed, which covered all the important research topics from algorithm-level neural network compression to hardware-level sparse-convolution-specific circuit design and architecture design space exploration (DSE).
- Two HW/SW partition strategies, which targeted different hardware settings, were proposed and corresponding designs were implemented on a Intel Arria 10 GX1150 FPGA. Experimental results have shown that our optimization scheme compressed the memory footprint of YOLOv2's CNN model by $20\times$ and the computational workload by $7\times$ at the cost of only 2.35% loss in detection accuracy, which had no obvious impact on the effectiveness of the real-world application tested in this work. The design that targeted at platforms with powerful CPUs can achieved a detection speed of 72.5 at 416×416 input resolution, while the design targeted at embedded platforms could also perform real-time detection at the frame rate of 61.9 fps. Our design shows more than $3.3\times$ improvement on throughput over the

best FPGA-based YOLO accelerator reported in the literature.

The paper is organized as follows: Section II first reviews the YOLOv2 object detection algorithm and the idea of the ABM-SpConv method. Section III describes the target heterogeneous platform, the HW/SW partition of YOLOv2 detection algorithm, and the design of the system. Section IV proposes the algorithm-level model optimize strategy that serves for hardware deployment. The implementation of the entire system and experimental results are shown in Section V. Section VI contains conclusions and further research directions.

II. BACKGROUND

A. REVIEW OF THE YOLOv2 ALGORITHM

YOLOv2 [5] is a classic one-stage object detection algorithm. Compared to traditional two-stage detection algorithms (such as R-CNN [31], Fast R-CNN [1], fasterrcnn [2], etc.), YOLOv2 directly converts the problem of bounding box positioning into an end-to-end regression solution. Since YOLOv2 avoids the process of generating hundreds of candidate boxes, the execution speed of the algorithm is significantly improved over the two-stage detection schemes, which makes YOLOv2 an excellent choice for implementing real-world applications.

Fig. 1 illustrates the whole detection flow of the YOLOv2 algorithm, which can be generally divided into three basic procedures: The first one is preprocessing of the input image. It involves resizing images of different input resolutions to a uniform size by using bilinear interpolation, and then subtracting the average brightness of all the images in the dataset to avoid distortion effect, such as over brightness.

The second procedure is the main CNN forward computation. In the YOLOv2 algorithm, the input image is divided into several grids according to the frame size, e.g., 13 × 13 grids for a 416 × 416 input image, and each grid is responsible for predicting five anchor boxes with different aspect ratio. Corresponding to each anchor box, the CNN outputs a 25-dimensional vector: one number for the probability the box contains an object, four numbers to represent the bounding box coordinates in relation to the anchor box, and 20 dimensional probability for each of the categories in the training dataset. In YOLOv2, the default CNN includes 23 convolutional layers, five max-pooling layers, one reorganization (Reorg) layer and one concatenation (Concat) layer. The activation layer of the network uses the Leaky-relu function with a negative slope of 0.1. Each convolutional layer is followed by a Batch Normalization (BN) layer, which is designed to avoid the problem of gradient vanishing during neural network training. In this work, we have slightly modified the YOLOv2 CNN to facilitate the hardware implementation. The original last layer *conv*23 was split into *conv*23 and *conv*24 to predict location and category information independently, which will be explained in detail in Section IV-D.

The final procedure is post-processing, which extracts the coordinates of the bounding box and the category information from the output of the CNN, and then filters them by performing non-maximum suppression (NMS) to get the best detection result. The bounding boxes of the objects are finally drawn and displayed for the user.

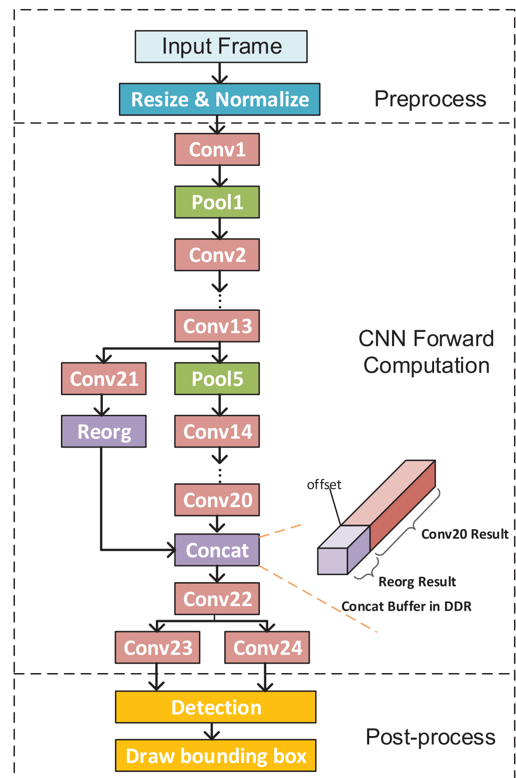


FIGURE 1. Processing flow of the YOLOv2 object detection algorithm. Leaky-relu activation and BN operation are applied after each convolution layer. The original *conv*23 layer is split into *conv*23 and *conv*24 to improve detection performance in this work.

We summarize some of the important computational characteristics of the YOLOv2 algorithm in Table 1, where #Param represents the total number of parameters of the CNN model and #FLOP represents the total amount of floating-point operations (each MAC operation contains one floating-point addition and one multiplication), which changes according to the input resolutions. Detection accuracy is measured in terms of mean Average Precision (mAP), and the scores shown in Table 1 are based on the PASCAL VOC 2007 dataset.

TABLE 1. Computational characteristics of the YOLOv2 algorithm.

Input Resolution	#Param (10^6)	#FLOP (10^9)	mAP
288 × 288	50.6	14.1	69
352 × 352		21.0	73.7
416 × 416		29.4	76.8
480 × 480		39.1	77.8
544 × 544		50.2	78.6

To help us make correct decisions on HW/SW partitioning, we have measured the detailed execution time of each procedure of the detection algorithm based on a software

implementation on CPU. The breakdown of the execution time is summarized in Fig. 2. It can be observed that the CNN forward computation consists of up to 90.2% of the total computational workload of the detection algorithm. Therefore, the key design challenge of implementing YOLOv2 on FPGA is to develop a dedicated hardware processor that can efficiently accelerate the forward computation procedure of CNN. Detailed discussion of hardware-software partition strategy will be presented in Section III-B.

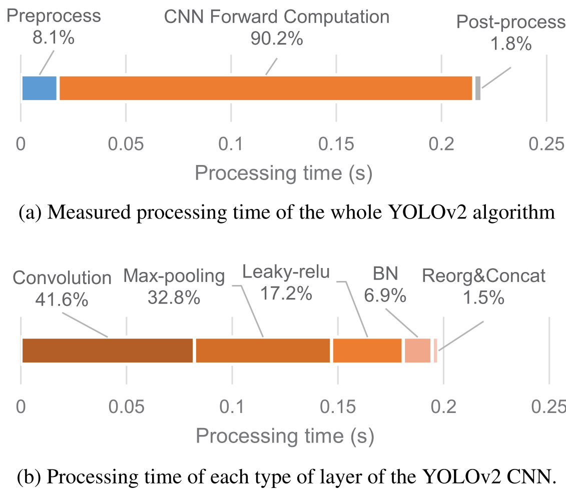


FIGURE 2. Breakdown of the execution time of each procedure of the YOLOv2 flow. An open-source design [32] based on Pytorch was used and the time was measured on an Intel i9-7960X CPU.

B. THE ABM-SpConv ALGORITHM

The ABM-SpConv algorithm is a special sparse convolution approach that targets at flexible and balanced resource utilization of the on-chip logic and DSP resources for FPGA accelerator [27]. The key idea of this scheme is to decouple the accumulate and multiply operations involved in the convolution computation into two separated stages so that the computational complexity of multiplication can be reduced to a much lower rate over accumulation by sharing unique quantized weight values. The major benefit is that it significantly relaxes the resource demand for multipliers (DSP units) when implementing sparse convolution computation on FPGA device.

Given the input image or feature-map IF of the size $W \times H \times C$ and the weight kernel WT of the size $K \times K' \times C \times M$, the convolution computation can be expressed by the following equation:

$$OF_{w',h',m} = \sum_{c=1}^{C} \sum_{k=1}^{K} \sum_{k'=1}^{K'} IF_{w' \times S+k, h' \times S+k', c} \times WT_{k,k',c,m} \quad (1)$$

where S denotes the stride of the convolution sliding window, and OF represents output feature-map, respectively. Assuming that after certain special algorithm-level optimization (such as weight quantization and clustering), the parameters of the weight are all quantized in fixed-point format, and there only exists Q unique quantization values for the weight in the m-th weight filter WT. By denoting these unique values as $\hat{WT}_j$, $(j = 1, \cdots, Q)$, one could transform Equation (1) into the following format:

$$\begin{aligned} OF_{w',h',m} &= \sum_{i=1}^{w_1} IF_1[i] \times \hat{WT}_1 + \cdots + \sum_{i=1}^{w_Q} IF_Q[i] \times \hat{WT}_Q \\ &= \sum_{j=1}^{Q} (\hat{WT}_j \times \sum_{i=1}^{w_j} IF_j[i]) \end{aligned} \quad (2)$$

where $IF_j[i]$'s represent all the input feature-map pixels that are multiplied by the same unique quantized weight value $\hat{WT}_j$ within a weight filter, and w_j denotes the specific number of the feature-map pixels that corresponds to that $\hat{WT}_j$. The new convolution formula of Equation (2) avoids redundant multiplications that are performed on the same unique weight value of $\hat{WT}_j$, and can significantly reduce the computational workload of the multiplication operation involved in convolution. Moreover, all the computations that corresponds to the zero-valued weight can also be bypassed by saving $\hat{WT}_j = 0$ as traditional sparse convolution schemes resulting in an overall workload reduction as discussed in Section I. Fig. 3 gives a more intuitive illustration on how ABM-SpConv is performed on a sparse weight of the size $3 \times 3 \times 3 \times 1$. The feature-map pixels that are multiplied by the parameters with the same quantization value are labeled by the same color.

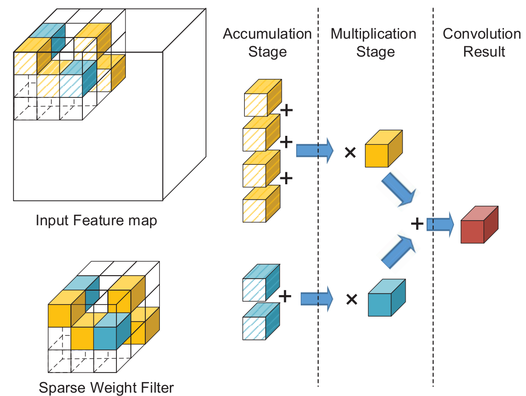


FIGURE 3. Computation flow of the ABM-SpConv scheme. Each cube represents a feature-map pixel or a non-zero valued parameter of the weight.

To achieve the maximum efficiency and highest throughput when implementing the ABM-SpConv scheme on FPGA, two design challenges related to computational workload and bandwidth have to be carefully addressed. Firstly, since CNN consists of multiple convolution layers and, in general, each layer has a different number of parameters which are also of different value distribution. This inconsistency can cause variable workload ratio between accumulation and multiplication operations among different weight filters and layers [27]. Secondly, in most of traditional neural network

pruning schemes [10], [33], the CNN model is pruned by using heuristic rules that only consider the salience of the parameters without taking into account of the computing and bandwidth characteristics of the underlying hardware architecture. When implemented on FPGA device, the unbalanced workloads and bandwidth will lead to low utilization of the hardware resource and undesired degradation of the overall system performance.

III. SYSTEM DESIGN

A. THE TARGET HETEROGENEOUS PLATFORM

In this paper, the proposed YOLOv2 accelerator targets at a heterogeneous computing platform which is composed of a general-purpose CPU and FPGA device. To facility the HW/SW co-design of the system, we adopt the OpenCL framework [34] to develop both the hardware and software partitions of the YOLOv2 accelerator as shown in Fig. 4. Hardware circuits, which implements the compute-intensive sub-algorithms, are first modeled in OpenCL codes in the form of kernel functions, and then compiled by using FPGA vendor's HLS compiler and mapped on the FPGA fabric, which is referred to as the OpenCL device. A C/C++ program executing on the CPU, which is defined as the host, provides vendor specific application programming interface (API) to control and communicate with the implemented OpenCL kernels. Software partition of the YOLOv2 algorithm is also executed on the host side. Data are transmitted between the CPU and FPGA through the PCIe link with very high throughput. Note that the CPU and FPGA device can also perform independent computation tasks asynchronously in parallel.

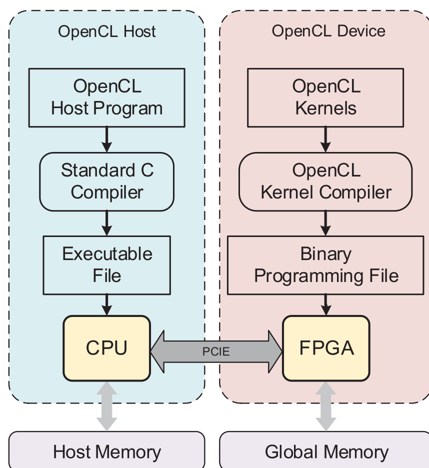


FIGURE 4. The target heterogeneous computing platform defined in the OpenCL framework.

B. HW/SW PARTITION

In this work, we propose two HW/SW partitioning strategies to target heterogeneous computing platforms with different computing capabilities.

The first strategy is for platforms with powerful CPUs, which can be used to allocate the workloads of a few non-performance-critical layers of the YOLOv2 network, such as max-pooling, Reorg and Concat. Then, all FPGA hardware resources can be used to accelerate the convolution layers to maximize the overall system performance. Fig. 5 illustrates the pipelining scheme of the computation tasks between the CPU and FPGA sides. In this work, the optimization goal is to maximize the computation throughput, i.e., the average number of frames that can be processed by the accelerator per second, we introduce inter-batch layer-wise pipelining to hide the execution time of the layer functions implemented on CPU. The pre- and post-processes are also arranged as software tasks in the pipeline at the beginning and the end of each frame. For object detection tasks on continuous image frames, like surveillance systems, when CPU can process the max-pooling and Reorg layers much faster than the convolution layers executed on FPGA, the average detection speed only depends on the execution time of convolution layers in the YOLOv2 algorithm.

The second partitioning strategy is for embedded or mobile platforms, such as the Xilinx Zynq FPGA device, in which CPUs are generally of low computing capacity. From the data shown in Fig. 2b, we can see that, even on a high-end x86 CPU, the processing time for max-pooling is as large as one-third of the whole CNN forward inference time, so it is highly possible that the max-pooling layers may consume more time than the FPGA-accelerated convolution layers. Therefore, for embedded devices, it is more reasonable to implement the whole YOLOv2 CNN in hardware circuits on FPGA fabric.

C. HARDWARE ARCHITECTURE DESIGN

One key design challenge of implementing sparse convolution algorithm in hardware is that the fine-grained unstructured pruning result can cause inconsistent amount of non-zero parameters, which are irregularly located in each weight filter, which will introduce unbalanced computational workload and prevent the accelerator from taking full advantage of the intrinsic parallelism of the convolution algorithm when implemented on the traditional hardware structure of globally synchronized MAC array [35], [36].

To address this issue, we propose in this paper an accelerator architecture based on asynchronously executed parallel computing cores, each of which is implemented in an OpenCL kernel running on the FPGA. Fig. 6 shows the top-level structure of the proposed FPGA accelerator for YOLOv2. The accelerator consists of a task scheduler, a pair of Data Load/Store Units (DLU/DSU), multiple Sparse Convolution Units (SCU) and interfaces that connect to the external DDR memory and PCIe bus. An alternative Max-Pooling Unit (MPU) is also designed to provide options to implementing the max-pooling layer on FPGAs. The task scheduler is responsible for launching and synchronizing the asynchronously executed SCUs and receiving controlling signals and feature-map data from/to the OpenCL host processor

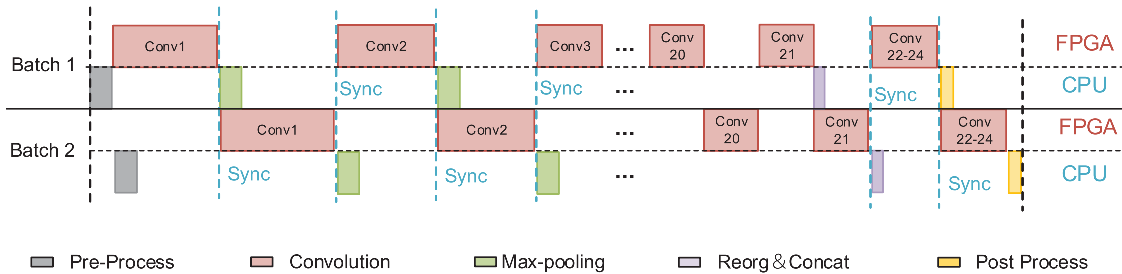


FIGURE 5. The proposed HW/SW partition scheme and processing pipeline of the YOLOv2 network on the CPU+FPGA heterogeneous platforms.

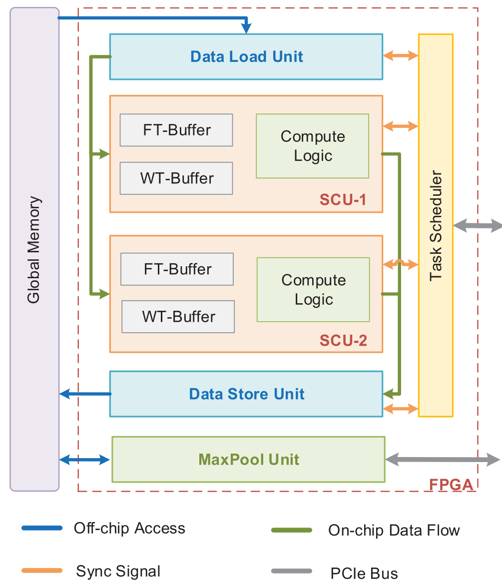


FIGURE 6. Overall architecture of designed on-chip system.

through the PCIe interconnection. The task scheduler periodically detects the status of each SCU and, whenever there is an idle one, it quickly launches a new computation task on that SCU and sends a command to the DLU/DSU to load required feature-map and weight data from external DDR memory to local buffers in each SCU. Then, the SCU starts to perform the convolution task independently for a relatively long period of time. To support imbalanced computational workload among different convolution tasks, each SCU kernel is designed its private local data buffers and loop counters such that frequent interrupt and synchronization by the task scheduler can be avoided to improve the efficiency of the proposed accelerator. The detailed design of the other kernel functions are described in detailed in the following sections. Table 2 lists all the hardware parameters that are defined for the proposed accelerator architecture.

1) DATA LOAD/STORE UNITS

The Data Load/Store Unit (DLU/DSU) is responsible for fetching the feature-map and weight data from the external DDR memory to each of the SCUs and storing the convolution results back to external memory. At the beginning of each layer convolution, the DLU first reads a prefetch window of feature-map data from external memory and broadcasts the data to each of the SCU simultaneously through the OpenCL kernel-to-kernel channels.

Then, the DLU loads the required weight filters corresponding to each of the convolution tasks to the SCUs in a round-robin way. After all the convolution tasks performed on the prefetch window are finished, the DLU returns to its initial state and waits a new command from the task scheduler. During convolution, synchronization of the SCUs is infrequently conducted only when all the convolution tasks on the same prefetch window are finished.

TABLE 2. Hardware parameters defined in the proposed architecture.

Parameter	Description
N_{in}	Batch size of the input
N_y	Number of parallel convolutions in the column direction
N_{scu}	Number of SCUs
N_{am}	Ratio of accumulators to multipliers in one SCU
D_f	Depth of line buffer in FT-Buffer
D_w	Depth of WT-Buffer
D_p	Depth of Pool-Buffer

2) SPARSE CONVOLUTION UNIT

SCUs are the main computing engine of the proposed accelerator. As shown by Fig. 7, each SCU is implemented as two autorun OpenCL kernels, namely the Accumulation Engine (AE) and Multiplication Engine (ME), which accelerates the accumulation and multiplication operations defined in Equation (2), respectively. Each AE has $N_{in} \times N_y$ parallel accumulators (adders). The Decoder is designed to select the required input feature-map pixels according to the nonzero weights. In each cycle, the Decoder reads the encoded non-zero weights sequentially from the Weight Buffer (WT-Buffer) and converts them into the physical address of the required feature-map pixels. Then the accumulator adds the input feature-map pixels $IF_j[i]$ which are loaded from the Feature-map Buffer (FT-Buffer) and increments the companion counter by one. After all the pixels that share the same $\hat{WT}_j$ have been accumulated, i.e., the counter reaches the bound w_j, the partial convolution result is sent into following FIFO. The accumulators in each AE are further

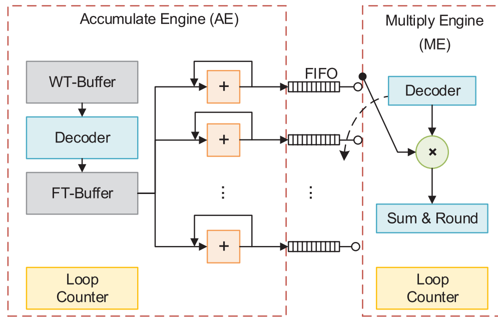


FIGURE 7. Internal structure of the proposed SCU.

divided into N_{am} groups and each group shares one multiplier located in the data-path. Therefore, each ME only has $N_{in} \times N_y/N_{am}$ multipliers. To balance the output and input data stream of the accumulators and multipliers in the pipeline, the multiplier is designed to periodically select one of the outputs of upstream FIFOs as its input operand in a round-robin manner.

Ideally, the design parameter N_{am} should match the proportional relationship between the computational complexity of the accumulate and multiplication operations for given pruned neural network model. A higher value of N_{am} can save a considerable amount of on-chip DSP resource, however, based on the previous experiment [27], it will cause a reduction in detection accuracy. Therefore, HW/SW co-optimization of this design parameter and the neural network pruning and quantization scheme should be conducted to find the most appropriate design point that can deliver the most balanced hardware cost and detection accuracy. We will address this issue in Section IV-B.

3) ON-CHIP BUFFER DESIGN

As shown by the left part of Fig. 8, the convolution operations are based on successive sliding windows, which normally share a large piece of overlapping area between each other. In this paper, we propose a line-buffer-based feature-map caching scheme to take advantage of this important feature of the convolution operation to improve the efficiency of external memory bandwidth. The internal structure of the FT-Buffer is illustrated on the right side of Fig. 8. For a convolution layer with the filter size of $K \times K' \times C$ and stride S, the FT-Buffer is designed with $N_y + ceil(K'/S) - 1$ two-port RAMs, forming a line-buffer structure with multiple read ports and a single write port. A prefetch window of the feature-map data is first read out from external memory and written into the FT-Buffer in a line-by-line way. Then, during convolution computation, the feature-map pixels correspond to N_y sliding windows along the column dimension are read out from N_y of the parallel output ports of FT-Buffer in a zigzag order simultaneously. To avoid possible memory read collisions, each line-buffer needs to store S lines of the prefetch window. In this way, N_y parallel convolutions along the column direction can be conducted in the same time. Along the row, successive convolutions are carried out sequentially, reusing the feature-map data stored in the FT-Buffer. The proposed line-buffer architecture can significantly reduce the utilization of the external memory bandwidth. For instance, given a prefetch window of $32 \times 32 \times 128$ pixels and a convolution kernel of $3 \times 3 \times 128$ with $S = 1$, the straightforward data fetching scheme needs to transfer 1.125M feature-map pixels from the external memory, whereas the proposed line-buffer-based approach only requires to fetch 128K pixels, which has reduced the bandwidth utilization by considerably $9\times$. The minimum size of the line buffer should be set to accommodate the feature-map used for the largest weight filter to perform convolution once.

Each SCU also has a WT-Buffer used to hold the pruned weight filters of CNN. Since there is a significant amount of zero-valued parameters which can be bypassed in the convolution operation, we propose to only store the quantized non-zero parameters and their coordinate information. Fig. 9 gives an example of how the pruned CNN model is encoded in this work. k, k', and c represent the coordinates of each non-zero parameter and are concatenated into a 16-bit word. The coordinates of those parameters that share the same quantized value are stored in successive positions followed by the corresponding fixed-point weight value (in 8-bit precision) and the total number (8-bit precision). As will see in Section V-B, the proposed sparse weight encoding approach can reduce the memory footprint of the original YOLOv2 model by $20.2\times$.

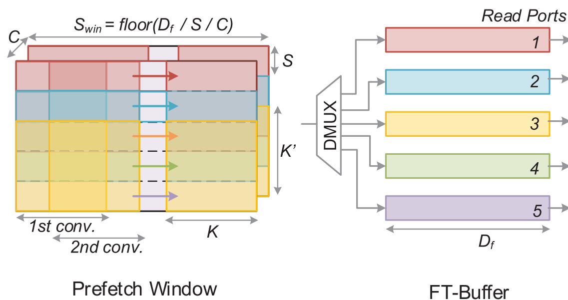
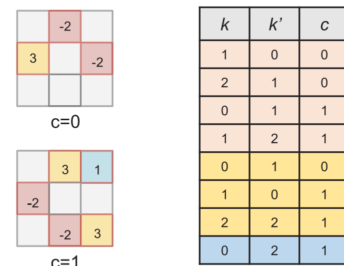


FIGURE 8. Design of on chip FT-Buffer. The picture shows the situation of $K = K' = 3, S = 1, C = 2$ and $N_y = 3$.



k	k'	c
1	0	0
2	1	0
0	1	1
1	2	1
0	1	0
1	0	1
2	2	1
0	2	1

FIGURE 9. An example of the proposed sparse weight encoding scheme.

4) HARDWARE MAX-POOLING UNIT

As shown in Fig. 10, the MPU is designed as an independent kernel outside of the pipeline to support flexible HW/SW partition strategies. When using FPGA to implement the max-pooling function, the MPU will be generated during the compilation stage, otherwise, binary programming file will not contain MPU logic. Considering that all max-pooling layers in YOLOv2 are based on 2×2 pooling window with $S = 2$, the proposed MPU uses one line-buffer (the Pool-Buffer) to cache half amount of the data in the pooling window. During execution, MPU first loads the odd-numbered rows of data from the global memory to the on-chip Pool-Buffer, and then reads the data of the even rows pixel by pixel and compares them with the one in the corresponding position in the Pool-Buffer in the same time. Finally, the larger partial results are stored into the registers, and when two intermediate results are both obtained, the largest value is selected as the output of the max-pooling operation and written back to global memory. Similar to the convolution operations, all N_{in} pooling windows in different batch items are processed in parallel to match the throughput of the SCUs.

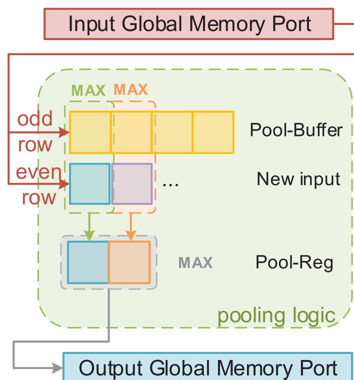


FIGURE 10. **Architecture of the proposed max-pooling unit.**

D. SOFTWARE FUNCTION DESIGN

1) SOFTWARE MAX-POOLING FUNCTION

The proposed software pooling implementation adopts Streaming SIMD Extensions (SSE) and POSIX Threads (Pthreads) to accelerate the max-pooling layer on CPU. As shown by Algorithm 1, P_a max-pooling threads are created in the channel direction of the feature-map, and each thread is responsible for performing an independent max-pooling operation. Within each thread, N_{in} pooling windows are divided into several groups, each of which combines the data from P_b pooling operations into a single vectorized data type, which can then be utilized in single instruction multiple data (SIMD) manner using SSE. When CPU is selected to implement the max-pooling layer, the multi-threaded max-pooling function will be executed asynchronously in parallel with the convolution units on FPGA as shown by Fig. 5.

Algorithm 1 Pseudo-Code of Software Max-Pooling

Load the feature-map pixels FT from global memory to host memory;
Set P_a, P_b;
$finish_thread_cnt = 0$;
Create & initialize mutex lock;
for *thread in* P_a **do**
  $start_channel = t \times (C/P_a)$;
  $end_channel = (t+1) \times (C/P_a)$;
  Create SSE type $SIMD_vec$ with P_b FT;
  $SIMD_group = N_{in}/P_b$;
  for *n from start_channel to end_channel* **do**
    for *f in all_pool_filters* **do**
      for *g in SIMD_group* **do**
        Load $SIMD_vec$;
        Perform max-pooling for current filter;
        Unzip $SIMD_vec$ and store back into global memory;
  Enable mutex lock;
  $finish_thread_cnt++$;
  Disable mutex lock;
Wait until $finish_thread_cnt == P_a$;

2) REORG AND CONCAT FUNCTIONS

As shown in Fig. 1, a Concat Buffer in global memory is designed to implement the Concat function in YOLOv2. Unlike other convolutional layers, after the accelerator calculates *conv*20, the result is stored back to the Concat Buffer with an offset. The result of the Reorg operation is also stored in the same buffer. In this way, *conv*22 can directly obtain the spliced data from the Concat Buffer without additional data movement.

E. PERFORMANCE MODELING

In order to analyze the actual throughput of the accelerator with respect to the theoretical peak performance from the perspective of HW/SW co-design, we adopt the concept of using the roofline model [37] to quantitatively guide the optimization flow. In this section, we first model two important performance metrics, including inference throughput and arithmetic intensity, which will be used in the roofline model to explore and find the best design point that can deliver the highest performance on the target platform.

As listed in Table 2, the proposed accelerator design is fully configurable with these parameters, providing a scalable architecture that can be easily modified to meet the performance and resources constraints of any FPGA device. For instance, the total number of accumulators consumed by the accelerator equals $N_{in} \times N_y \times N_{scu}$, while the total number of multipliers used are $N_{in} \times N_y \times N_{scu}/N_{am}$. All the algorithm-related parameters are defined in Table 3 as well.

TABLE 3. Parameters defined in algorithm optimization flow.

Parameter	Description
PR_j	Pruning rate for layer j
BW	Bit width of quantized weight
CL_j	Number of clusters for layer j
I_j	Arithmetic intensity of layer j
Sam_j	Ratio of add. and mult. operations of layer j

In the following discussion, we define P_j and O_{acc_j} as the number of parameters (model size) and accumulation operations (workload) of the j-th convolutional layer in the original YOLOv2 CNN model, while P'_j and O'_{acc_j} as the model size and workload of the j-th convolution layer after being pruned and quantized, respectively. The reason why we use the amount of accumulation operations to measure the workload is that the adders and multipliers in the SCU work in parallel in a pipelined manner, and the amount of multiplication calculation is much lower than accumulation, so the performance of the accelerator is closely related to the accumulation workload.

1) COMPUTATION THROUGHPUT

According to the behavior of sliding-window-based convolution operation, if the input feature-map is of the size $W \times H \times C$, the weight filter of this layer is of the size $K \times K' \times C$ and there are totally M filters in one convolution layer, then for the convolution of stride S and padding PD, the size of the output feature-map is $W' \times H' \times M$, which can be calculated by

$$W' = [(W + 2 * PD - K)/S + 1] \quad (3)$$
$$H' = [(H + 2 * PD - K')/S + 1] \quad (4)$$

Therefore, the size of the pruned model in the current layer equals to

$$P'_j = P_j \times (1 - PR_j) = K \times K' \times C \times M \times (1 - PR_j) \quad (5)$$

In the proposed architecture, the total workload of accumulation is

$$O'_{acc_j} = O_{acc_j} \times (1 - PR_j) = W' \times H' \times P_j \times (1 - PR_j) \quad (6)$$

Assuming that the accelerator works at a frequency of $Freq$, and in each clock cycle, the accelerator can effectively perform $N_{in} \times N_y \times N_{scu}$ accumulations, then the theoretical average processing time of the j-th layer per input frame can be estimated by

$$T_j = \frac{O'_{acc_j}}{N_{in} \times N_y \times N_{scu} \times Freq} \quad (7)$$

2) ARITHMETIC INTENSITY

In this paper, since the ME in the SCU does not involve off-chip memory access, arithmetic intensity is calculated as the proportional ratio of the number of accumulation operations to the actual amount of data fetched from external DDR memory during convolution. In the proposed YOLOv2 accelerator, the external memory bandwidth is spent on two types of data: feature-map (H_{f_j}) and encoded sparse weight (H_{w_j}). As discussed in previous section, in each layer, the whole input feature-map is processed after $G_{x_j} \times G_{y_j}$ times of prefeching, where G_{x_j} and G_{y_j} equal

$$G_{x_j} = ceil(\frac{W'}{floor[(S_{win} - K)/S] + 1}) \quad (8)$$
$$G_{y_j} = ceil(\frac{H'}{N_y}) \quad (9)$$

where S_{win} is the width of the prefetch window constrained by the depth of the feature-map buffer as follow

$$S_{win} = floor(\frac{D_f}{C \times S}) \quad (10)$$

Therefore, if each parameter is quantized to BW-bit word-length, the total amount (Byte) of feature-map data fetched in the current layer is

$$H_{f_j} = G_{x_j} \times G_{y_j} \times S_{win} \times [(N_y - 1) \times S + K'] \times C \times BW/8 \quad (11)$$

Normally, the number of the unique weight quantization values is marginal to the total number of parameter coordinates. Therefore, we only count the total number of transferred coordinates (16-bit word-length, 2 Byte) to estimate the average bandwidth per image frame:

$$H_{w_j} = \frac{G_{x_j} \times G_{y_j} \times P'_j \times 2}{N_{in}} \quad (12)$$

Finally, the arithmetic intensity I_j (OP/Byte) defined in the roofline model can be calculated by

$$I_j = \frac{O'_{acc_j}}{H_{f_j} + H_{w_j}} = \frac{O_{acc_j}}{H_{f_j}/(1 - PR_j) + (2 \times G_{x_j} \times G_{y_j} \times P_j)/N_{in}} \quad (13)$$

IV. HARDWARE-AWARE ALGORITHM-LEVEL OPTIMIZATION

A. THE OPTIMIZATION FLOW

As shown by Fig.11, the proposed algorithm-level optimization flow consists of three steps. The first step is CNN model pruning, which targets at balancing the reduction rates in both model size (memory footprint) and computational workload so that the performance gain delivered by the accelerator can be maximized without being limited by external memory bandwidth. The second step is to optimize the distribution of the value of the parameters so that the proportional ratio of the number of accumulate operation to multiplication in the sparse convolution algorithm matches the hardware parameter N_{am}. This step ensures that the average execution time of the ME is always shorter than that of the AE.

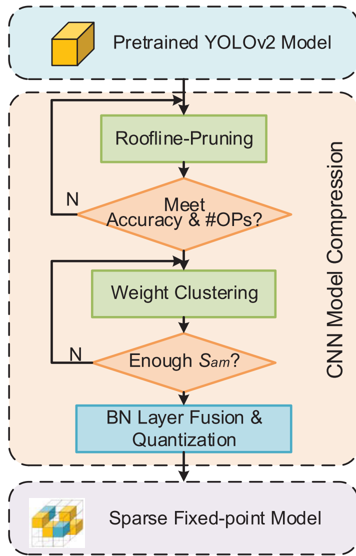


FIGURE 11. The proposed algorithm optimization flow.

The final optimization step fuses the BN layer into the convolution layer to avoid complicated floating-point operations required by batch normalization, and then quantize the whole CNN model in fixed-point format with shorter word-length to further reduce the memory footprint of the weight and implementation cost of the hardware circuits.

B. ROOFLINE-MODEL-BASED WEIGHT PRUNING

As explained by the roofline model theory [37], the highest attainable performance of a certain hardware processor may be bounded by either the on-chip computational resource or external memory bandwidth. A typical roofline model is illustrated in Fig. 12a. When the implemented algorithm has low arithmetic intensity, it will fall into the memory-bound area of the roofline model, which means that the external memory interface can not provide sufficiently large data stream to meet the throughput requirement of the computation units in the processor. As the arithmetic intensity increase, the upper limit of the highest attainable performance improves accordingly until it reaches the compute-bound area, which indicates that the memory bandwidth is always sufficient and the peak performance of the accelerator is limited by the on-chip hardware resources only. It can be inferred from Equation (13) that the arithmetic intensity I_j decreases accordingly as the pruning rate PR_j increases. Therefore, in theory, there is a maximum value for PR_j which can guarantee the accelerator fall into or near the optimal design point that is located at the upper left corner of the roofline model.

Therefore, we propose a new roofline-model-based pruning algorithm (also referred to as roofline-pruning). The goal of roofline-pruning is to make the arithmetic intensity of each convolution layer as close as possible to the junction of the memory-bound area and the computation-bound area. Thus, the highest attainable performance of the hardware accelerator can be obtained.

It can be derived from Equation (13) that, for the j-th convolution layer, the ideal pruning rate PR_j is

$$PR_j = 1 - \frac{I_j \times N_{in} \times H_{f_j}}{N_{in} \times O_{acc_j} - 2 \times I_j \times G_{x_j} \times G_{y_j} \times P_j} \quad (14)$$

Note that for some layers whose original arithmetic intensity is very low (such as layers with 1×1 weight filter size), the pruning rate calculated by the Equation (14) might be negative. In this case, we will directly set the pruning rate of this layer to zero. During CNN model pruning, we adopt a heuristic criteria, which is similar to [33], to measure the importance of each parameter and accumulate the saliency score over the whole dataset during retraining to obtain a more accurate threshold for weight trimming in each layer than that of [33]. At the same time, in order to satisfy the limitation of the fixed buffer depth of WT-Buffer, we also constrain the maximum number of non-zero values in each weight filter during pruning, which is done by automatically counting the occurrence of the non-zero weight in each weight filter and rounding number to the upper nearest power of two by trimming an extra small amount of weights. Based on our experiment on YOLOv2, the smallest weight buffer depth D_w that does not introduce obvious accuracy drop is 1024.

C. WEIGHT CLUSTERING

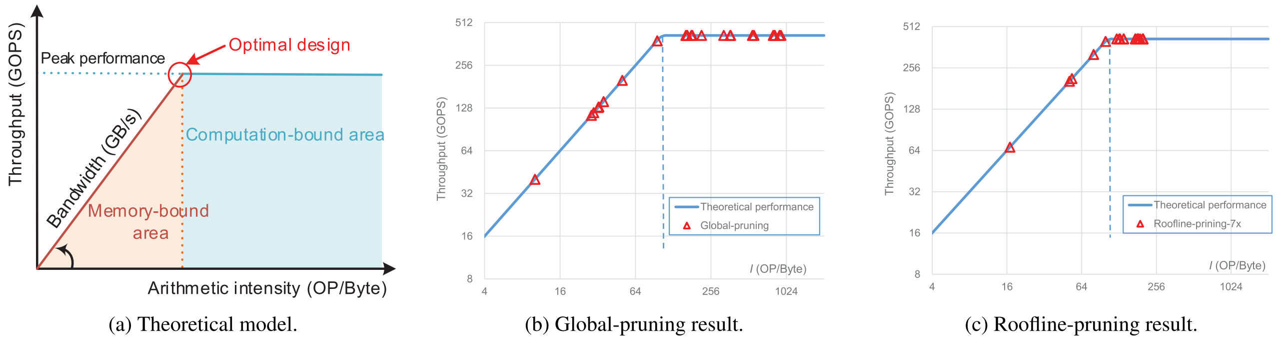
As been discussed in Section III-C, the proposed SCU consists of two independent accumulator and multiplier arrays, and the number of the accumulator is N_{am} times the number of the multiplier. Therefore, for a given hardware architecture, it is desired that the proportional relationship between the addition and multiplication operations of the convolution conducted in the format of ABM-SpConv algorithm should match the hardware parameter N_{am}. This workload reduction ratio can be quantitatively calculated by

$$S_{am_j} = \frac{P'_j}{\sum_{m=1}^{M} Q_{m_j}} \quad (15)$$

where Q_{m_j} is the number of unique non-zero parameters in the m-th weight filter.

Since P'_j is fixed after the CNN model has been pruned, S_{am_j} will be solely determined by the sum value of Q_{m_j}. Unfortunately, the distribution of the unique weight values in each layer of the non-structurally pruned CNN model is normally random. If the pruned weight filter results in a large $\sum_{m=1}^{M} Q_{m_j}$ causing $S_{am} < N_{am}$, the actual execution time of the implemented ABM-SpConv convolution will be longer than the theoretical one due to the gap between the workload of the multiplication operator and the limited number of hardware multipliers.

To avoid possible mismatch between S_{am} and the hardware parameter N_{am}, we propose to perform clustering of the value of the parameters in each weight filter before quantization to strictly limit the number of unique non-zero parameters of each weight filter. Similar to the work of [33], K-means



(a) Theoretical model. (b) Global-pruning result. (c) Roofline-pruning result.

FIGURE 12. **Effects of the proposed roofline-pruning scheme. (a) Theoretical roofline model. (b) The pruning result using the scheme presented by [10].Pruning rate was set to 89.9% and the achieved reduction rate on workload was 2.8x. (c) Pruning result using our rooline-pruning method, pruning rate is 90.1% and the workload reduction ratio is 7.0×. The data is measured on an Intel Arria 10 GX1150 FPGA with hardware paramter setting of $N_{in} = 40$, $N_y = 13$, $D_f = 3840$, $D_w = 1024$, $BW = 8$.**

algorithm is adopted to conduct weight clustering and we have also introduced BIRCH [38] algorithm to fine-tune the clustering result to improve the detection accuracy. For each convolution layer, given the pruning rate PR_j obtained from the pruning step, the flow first calculates the minimal value for $\sum_{m=1}^{M} Q_{m_j}$ that satisfies $S_{am} > N_{am}$, and then set $\sum_{m=1}^{M} Q_{m_j}/M$ as the average number of categories for each weight filter. After weight clustering, the detection accuracy is measured again, and if the accuracy drop is within the pre-set budget, then the flow continues to perform quantization. Otherwise, the number of categories is slightly adjusted and parameters are clustered again until the detection accuracy goal is met.

D. BN FUSION AND DYNAMIC FIXED-POINT QUANTIZATION

Equation (16) shows the core function performed by the BN layer in YOLOv2. y_{conv} and y_{bn} represent the input and output of BN layers, respectively. μ and σ^2 denote the running mean and variance of mini-batch and ϵ is a very small constant to prevent the denominator from being zero. The BN operation requires complicated floating-point arithmetic operations which will consume significant amount of logic and DSP resources. Scaling factor γ and bias β are trained parameters.

$$y_{bn} = \gamma \times (\frac{y_{conv} - \mu}{\sqrt{\sigma^2 + \epsilon}}) + \beta \tag{16}$$

In this work, we adopt the BN fusion strategy from [39] to transform Equation (16) into the form of $y = \alpha \times x + \eta$, where

$$\alpha = \frac{\gamma}{\sqrt{\sigma^2 + \epsilon}}, \quad \eta = \beta - \frac{\mu\gamma}{\sqrt{\sigma^2 + \epsilon}} \tag{17}$$

Thus, the BN operation can be merged with convolution by scaling the weight as follow

$$\bar{WT}_{k,k',c,m} = WT_{k,k',c,m} \times \alpha \tag{18}$$

and adding η to Equation (1) as a bias.

In FPGA designs, converting full precision floating-point arithmetic operation to lower-bit fixed-point computation can save a considerable amount of on-chip hardware resource and external memory bandwidth. In the proposed optimization flow, we extend the strategy introduced by [13] to perform quantization on the weight and bias of the YOLOv2's CNN, as well as the input and output feature-map of each convolutional layer. The quantized data are presented in the following format:

$$n = (-1)^s \times 2^{-FL} \sum_{i=0}^{BW-2} 2^i \times B_i \tag{19}$$

where s is the sign bit, B_i denotes the mantissa value of the i-th digit and BW represents the bit-width of the quantized number n. During quantization, we first set BW to the desired value, and then dynamically traverse all possible values of FL within a predefined interval and measure the detection accuracy of the quantized YOLOv2 network in a layer-by-layer manner. The FL that corresponds to the highest accuracy is selected as the optimal quantization setting.

One problem that should be carefully handled is that the last layer of the original YOLOv2 network has 125 output channels, in which 105 of the channels are used for predicting the confidence and the probability of each category, while the other 20 channels are utilized for predicting the coordinates of the bounding box. Normally, it is impossible to find a single unified FL for the entire layer without significant information loss since the distribution of the values of these two groups of channels is very different. To address this issue, we spilt the original last layer into two layers, i.e., $conv23$ which has 20 channels for predicting location information, and $conv24$ which has 105 channels for predicting object category information as shown by Fig. 1. Then, quantization can be separately conducted for these two layers.

V. EXPERIMENT AND RESULTS

A. EXPERIMENTAL SETUP

To evaluate the effectiveness of the proposed optimization scheme and the efficiency of the proposed hardware accelerator, we have implemented the final design on the Intel A10 FPGA Development Kit with an

TABLE 4. Comparison between different pruning methods [a].

Method	Pruning Rate	#Param (10^6)	#OP (10^9)	#OP Reduction	mAP	Avg. I (GOP/Byte) [b]
Baseline YOLOv2	-	50.6	29.4	-	76.8	-
Global-pruning [10]	89.9%	5.1	10.5	2.8×	75.55	318.3
	92.0%	4.1	9.6	3.1×	74.57	282.3
	95.1%	**2.5**	8.0	3.7×	71.87	217.6
Roofline-pruning-5x	89.2%	5.4	5.8	5.1×	**75.63**	172.2
Roofline-pruning-6x	90.3%	4.9	4.5	6.5×	75.28	142.4
Roofline-pruning-7x	90.1%	5.0	**4.2**	**7.0**×	74.45	**127.4**

[a] The experimental results are based on constraining the maximum number of non-zero values in each weight filter to 1024.
[b] The shown arithmetic intensity I is calculated by $N_{in} = 40, N_y = 13, D_f = 3840, BW = 8$.

Arria-10 GX1150 FPGA on board. The FPGA board was installed on a workstation equipped with an Intel i9-9900k CPU and 64GB of memory. The software development tools used were Pytorch v1.0 and Intel FPGA OpenCL SDK v19.3.

In the design of OpenCL code, we set DLU, DSU and MPU as single-thread kernel, and pass functinoal parameters through OpenCL host. SCUs are configured as autorun kernels and transmit parameter through OpenCL kernel-to-kernel channel. Multiple parallel computing SCUs are generated by using the *__attribute__((num_compute_units(N_{scu})))* pragmas. The generation of parallel computing circuits in each SCU is realized by loop unrolling using the *#pragma unroll* directive.

B. ALGORITHM OPTIMIZATION RESULTS

We have integrated the proposed pruning, clustering, and quantization algorithms into an end-to-end CNN model optimization flow by using Pytorch. The network pruning results are reported in Table 4, where #Param and #OP represent the remaining amount of parameters and workload of the pruned CNN model, and Avg.I_j represents the average arithmetic intensity of each layer. The detection accuracy was measured in terms of mAP. To demonstrate the effectiveness of the proposed schemes, we also report the pruning results that are obtained by using the scheme presented by [10].

From the perspective of reducing CNN's workload, it can be seen from Table 4 that the proposed roofline-pruning approach can reduce the computational workload of the YOLOv2 algorithm by up to 5.1× at the cost of only 1% loss in accuracy, while the scheme of [10] can only achieve a reduction rate of 2.8×. When the constraint on accuracy loss is relaxed to 2%, our method can reach a considerable 7× reduction ratio, which is almost 2× higher than that of [10]. From the perspective of roofline model, the proposed pruning scheme successfully moves all the layers to the places that are close to the optimal point as shown in Fig. 12c. Although four layers, *conv*4, *conv*21, *conv*23 and *conv*24, are still in the memory-bound area, their throughput have been significantly improved. Based on our prior experience, the two points accuracy drop has no obvious impact on the effectiveness of the real-world application, so we choose the model with #OP reduction of 7× (i.e., the roofline-pruning-7x model) as the candidate model for subsequent optimizations.

Weight clustering results are reported in Fig. 13 and Table 5. In the experiment, we first performed several rounds of clustering by using a single class number for all layers. As the results in Fig. 13 shows, it was found that, for most layers, a number of 16 clustering classes can guarantee a sufficiently large reduction rate in multiplication workload (i.e., $S_{am_j} > 8$), with exceptions for the first and second layers. Secondly, we fine-tuned the clustering result by setting independent class numbers for the first and second layers, respectively Table 5 reports several important fine-tuning results that were used to explore the best trade-offs between S_{am_j} and the detection accuracy. For instance, in the first column of the table, the term "8+16+16" refers to setting of class number of 8, 16 and 16 for the first, second and the remaining layers, respectively. S_{am_1} and S_{am_2} represent the adjusted reduction rate for the first and second layers. It is found that decreasing the class number of the second layer to 8 can enlarge S_{am_2} to 10 with only one point drop in detection accuracy. However, S_{am_1} could not be satisfactorily adjusted to a reasonable value without affecting the accuracy. Therefore, the combination of "6+8+16" is selected as the optimal clustering result.

TABLE 5. Experimental result on weight clustering.

Cluster Method	S_{am_1}	S_{am_2}	Avg. S_{am_j}	mAP
-	1.0	1.0	1.0	74.54
32+32+32	1.0	3.6	12.2	73.45
16+16+16	1.4	5.4	25.3	74.03
8 + 8 + 8	2.7	10.8	50.6	71.10
8 +16+16	2.7	5.4	25.4	73.56
6 + 8 + 8	3.6	10.8	**50.7**	70.17
6 + 8 +16	3.6	**10.8**	25.6	**73.63**
5 + 8 +16	**4.3**	10.8	25.6	70.00

Previous studies [22], [25] have shown that 8-bit world-length is sufficient for the YOLO series object detection algorithm to maintain a reasonable accuracy loss less than 1%. Therefore, we also select $BW = 8$ to quantize the network in the final implementation. For the Leaky-relu layer, the original negative slope parameter with the value of 0.1 was also quantized in 8-bit fixed-point with $FL = 10$. During quantization, several rounds of fine-tuning of the quantized neural network were conducted, which slightly improved the final detection accuracy from 73.63 to 74.45.

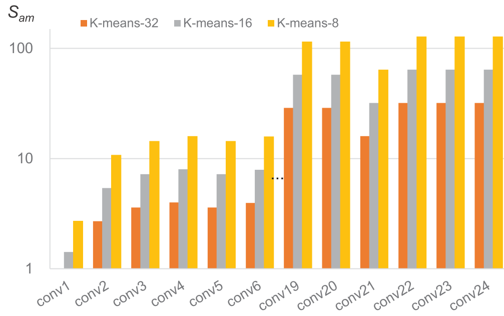


FIGURE 13. Comparison of the clustering results with different numbers of class. Due to space limitations, only a few selected layers are shown.

In Table 6, we summarize all the algorithm optimization results and compare them with the original YOLOv2 algorithm.

TABLE 6. Summary of the algorithm optimization results.

Model	YOLOv2 baseline [5]	Sparse-YOLO (ours)
Pruning Rate	-	90.1%
Cluster Setting	-	6+8+16
BN Fusion	-	✓
Model Precision	32-bit	8-bit
Workload (GOPs)	29.4	4.2
Model Size (MByte)	202.4	10.0 [a]
mAP	76.8	74.45

[a] The reported model size is the result after sparse weight encoding.

C. DESIGN SPACE EXPLORATION

An end-to-end design space explore (DSE) flow is designed as Fig. 14 shows. The proposed DSE flow is implemented as a python script, which can automatically launch the OpenCL SDK to perform multiple rounds of faster compilation of the kernel codes, then collect and analyze the hardware resource utilization data extracted from the compilation report.

Firstly, according to the values of the dimensional parameters of each convolution layer of the YOLOv2 CNN, the flow finds that the largest common divisor for the parameter H is 13, which means that setting the degree of parallelism in the column convolution direction to 13 can guarantee a full utilization of the parallel convolution pipelines for all layers. Therefore, in the final design, the DSE flow chooses this common divisor as the optimal value for N_y.

Next, the flow automatically compiles the designed kernel codes with a set of predefined values for the hardware parameters N_{in} and N_{scu}. The results are shown as the colored dots in Fig. 15. The average execution time per frame is calculated by using the performance model defined in Section III-E. Then, the remaining hardware resource and performance information are estimated by using linear-regression techniques, and the whole design space is finally established. The FPGA on-chip hardware resource constrains, including logic, memory and DSP blocks, are illustrated in black dotted lines. Since the maximum bit width of the DDR controller on

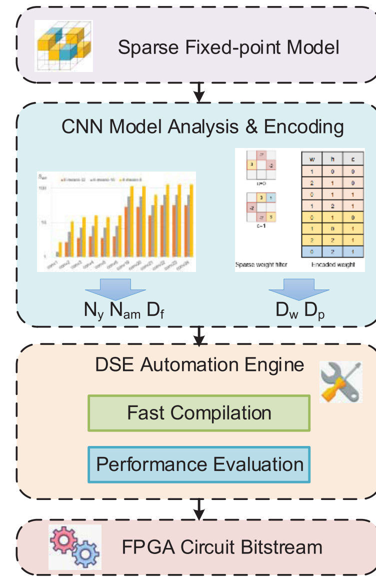


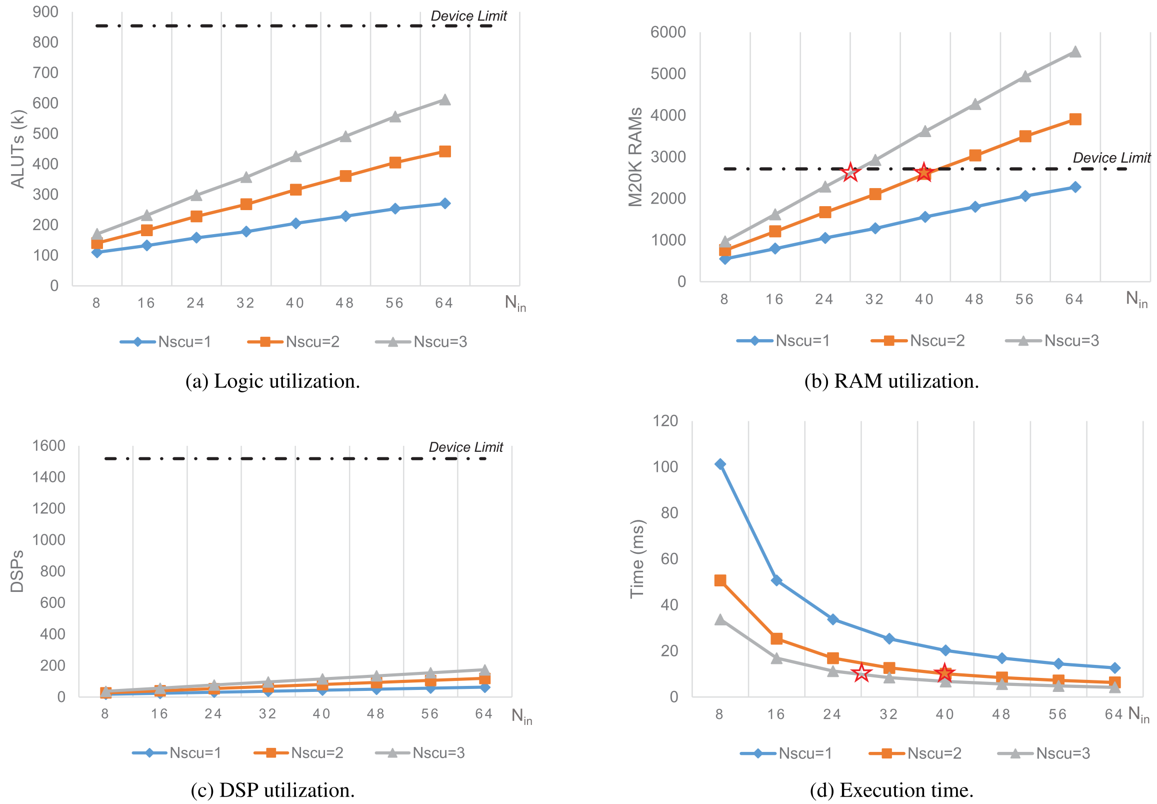
FIGURE 14. Design space exploration flow.

Arria-10 GX1150 FPGA is 512-bit, a largest batch size of 64 can saturate the external memory bandwidth. Therefore, N_{in} was only explored from 8 to 64. The depth of the line buffer in FT-Buffer D_f was set to meet the minimum requirements of the YOLOv2 network as $3 \times 1 \times 1280 = 3840$ for *conv*22. In addition, D_w was set to 1024 according to the constraint applied in the pruning stage.

From the results shown in Fig. 15, we can see that two design points that have the parameter setting of $N_{in} = 28$, $N_{scu} = 3$, and $N_{in} = 40$, $N_{scu} = 2$ can achieve the highest theoretical performance within the device's hardware resource limit. Since in the FPGA bitstream compilation stage, high logic and memory resource utilization (usually when the ratio is larger than 80%) may lead to failures in placement and routing or large degradation in operating frequency, both of these two design points were reserved as optimal design candidates for the final compilation. The performance and hardware parameters of the final selected designs are reported in Table 7. The performance data is based on the average

TABLE 7. Summary of the design space exploration results.

Implementation	Sparse-YOLO-1 (CPU max-pooling)	Sparse-YOLO-2 (FPGA max-pooling)
N_{in}	40	40
N_y	13	13
N_{scu}	2	2
N_{am}	8	8
D_f	3840	3840
D_w	1024	1024
D_p	-	52
Frequency (MHz)	211	204
Throughput (GOPS)	2129.7	1817.5
Throughput (fps)	72.5	61.9



(a) Logic utilization.

(b) RAM utilization.

(c) DSP utilization.

(d) Execution time.

FIGURE 15. Design space exploration results for the hardware parameters N_{in} and N_{scu}. The values of other hardware parameters are $N_y = 13$, $N_{am} = 8$, $D_f = 3840$, $D_w = 1024$. The execution time is estimated using performance model when frequency is 200MHz.

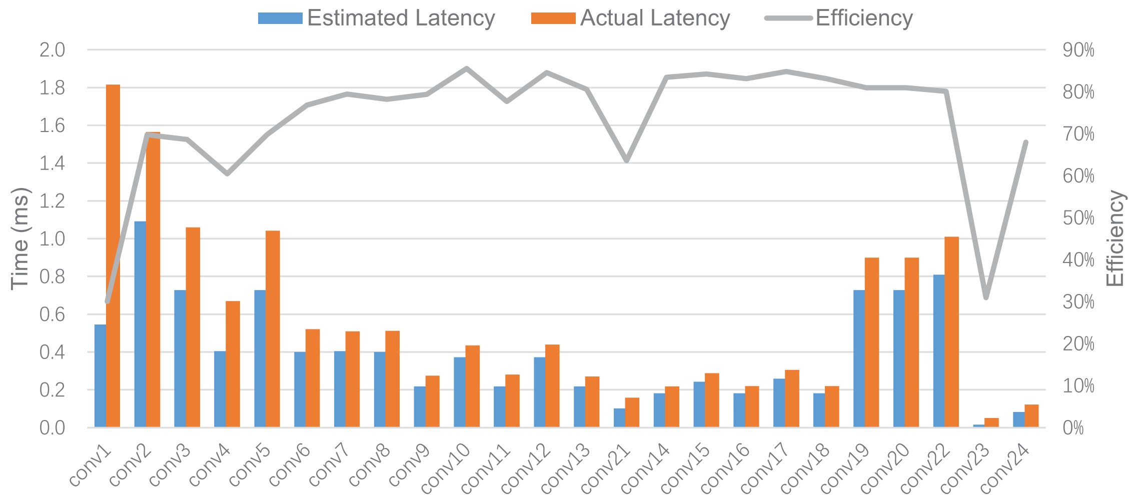


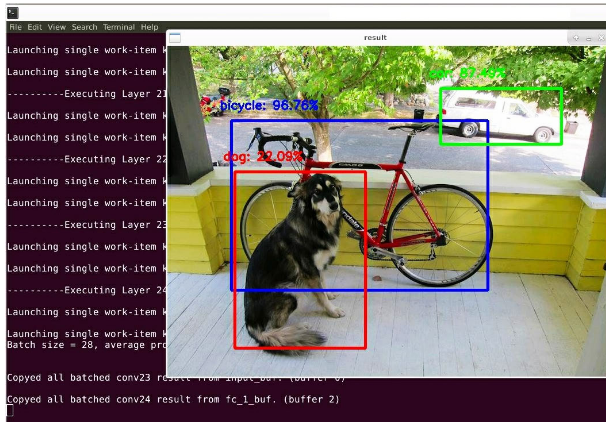
FIGURE 16. Comparison of the efficiency of the convolution computation for each layer of the YOLOv2 CNN. The estimated time is calculated by using the theoretical performance model, and the actual time is on the Sparse-YOLO-1 design.

processing time of one thousand pictures selected from the PASCAL VOC dataset as shown by Fig. 17a.

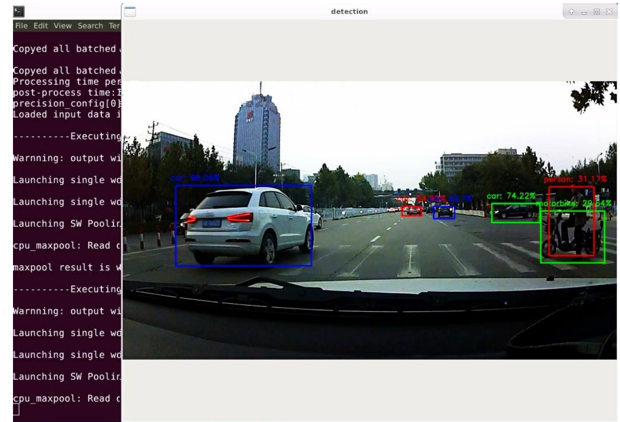
Two designs with different HW/SW partition strategies have been evaluated. Sparse-YOLO-1 only implements the convolution layer on FPGA, whereas Sparse-YOLO-2 also implements the max-pooling layer on FPGA. The experimental results have shown that the max-pooling time in Sparse-YOLO-1 can be well hidden by the convolution computation by using the proposed task pipelining scheme shown in Fig. 5.

D. EFFICIENCY AND DETECTION RESULTS

In Fig.16, we compare the actual execution time of each convolution layer with the theoretical execution time obtained



(a) Detection result on the PASCAL VOC dataset (Sparse-YOLO-1).



(b) Real-time detection of vehicle and pedestrian on a autonomous driving FPGA platform (Sparse-YOLO-2).

FIGURE 17. **The implemented Sparse-YOLO detector running on the FPGA platform.**

TABLE 8. **Comparison with the baseline GPU implementation and state-of-the-art FPGA accelerators.**

Implementation	Resolution	CNN Precision	Target Device	FPGA Resource Usage				Throughput[a] (fps)	Accuracy[b] (mAP)	Power[c] (W)
				Fmax	Logic	DSP	RAM			
GPU-Baseline [5]	416 × 416	FP32	NVIDIA Titan X	-	-	-	-	67	76.8	227
OpenCL-YOLO [22]	416 × 416	Int8	Arria-10 GX1150	200MHz	34%	72%	68%	18.9	76	26
FCLCNN [25]	416 × 416	FP16	Arria-10 GX1150	278MHz	50%	98%	88%	18.6	76.08	45
Lightweight YOLOv2 [18]	224 × 224	Binary	Xilnx ZU9EG	300MHz	49%	15%	93%	40.8	67.6	4.5
Sparse-YOLO-1	416 × 416	Int8	Arria-10 GX1150	211MHz	44%	5%	95%	72.5	74.45	26
Sparse-YOLO-2	416 × 416	Int8	Arria-10 GX1150	204MHz	45%	6%	96%	61.9	74.45	26

[a] Throughput is to measured in terms of the average processing time of 1000 frames after 200 frames warm up.
[b] Accuracy is measured by testing the entire PASCAL VOC2007 dataset.
[c] Power consumption is measured by using a power meter connected to the FPGA board.

by using the performance model of Equation (7). It can be seen from the data that, for majority of the convolution layers in YOLOv2, the execution efficiency of the proposed FPGA accelerator is between 70% $\sim$ 80%. There are two special cases in which the execution efficiency is relatively low. The first case is the four layers that have relatively low arithmetic intensity (including $conv4$, $conv21$ $conv23$ and $conv24$ as shown in Fig. 12c), and fall into the memory-bound area of the roofline model. In this case, the processing time is determined by the data transmission time, which means that FPGA on-chip computing units often stop to wait for the required data to be fetched. Another case is the first layer whose S_{am} is smaller than the value of the hardware parameter N_{am}. As been discussed in Section IV-C, this gap causes frequent pipeline stalls of the accumulator array to wait for the multiplier array in the SCU, resulting in increased convolution time.

To demonstrate the effectiveness of the proposed YOLOv2 accelerator, we have also implemented a real-world application of vehicle and pedestrian detection on a autonomous driving experiment platform as Fig. 17b shows. Our design can perform real-time object detection (frame rate $>$ 60 fps) on the video stream captured by a high-definition resolution camera.

E. COMPARISON WITH STATE-OF-THE-ART

Table 8 compares our design with the baseline software implementations of YOLOv2 on a Nvidia Titan X GPU [5] and three state-of-the-art FPGA-based accelerators. Each of the reference FPGA design adopts a different type of convolution strategy, including spatial domain convolution [22], frequency domain convolution (Winograd) [25] and multiplication-free binary convolution [18]. All reference designs are based on the same backbone network, i.e., DarkNet-19. The first two reference designs share the same input resolution of 416 × 416 with our design, while the work of [18] uses an input resolution of 224 × 224. Therefore, the computational workload of the CNN used in [18] is only 29% of that of the proposed and the other two reference designs. Moreover, both [25] and the

Sparse-YOLO-1 implementation deployed the max-pooling function on CPU, while [18], [22] and Sparse-YOLO-2 implemented hardware pooling circuit on FPGA. All designs have implemented the Reorg and Concat layers on CPU.

Compared to the baseline YOLOv2 implemented on GPU, our Sparse-YOLO-1 design achieves an 8% higher throughput at the cost of two points drop in detection accuracy. The most significant advantage of our scheme is the 8.7× reduction in energy consumption. The high energy efficiency makes the proposed accelerator an ideal choice for implementing the YOLOv2 detection algorithm for Internet of Things (IoT) or edge computing applications. When comparing to [22] and [25], both of which are based on the same FPGA device and have very similar performance, the two implementations of the proposed accelerator improve the detection throughput by 3.8× and 3.3×, respectively. Due to the effectiveness of the ABM-SpConv approach, our design saves a significant amount of on-chip DSP blocks, which avoids the hazard of competition for hardware resource when integrating other functional units, such as image denoising, with the the proposed YOLOv2 accelerator on a single FPGA device. Although the design of [18] works on a much smaller input resolution, which has reduced the computational workload to nearly one-third, our scheme still shows a more than 50% advantage in performance and 2.7× reduction in DSP usage. The main reason that our design consumes more power is because the design of [18] is based on a 16nm FPGA, while the device used in this work is based on 20nm technology.

VI. CONCLUSION AND FUTURE WORKS

In this paper, for the first time, fine-grained unstructured pruning was applied to an object detection task and proved to be very effective.

We have presented a high-throughput YOLOv2 accelerator design on a CPU+FPGA heterogeneous platform based on OpenCL framework. The asynchronously executed parallel convolution cores are designed to address the issue of imbalance workloads and bandwidths. We have also proposed a HW/SW co-design methodology including hardware-aware neural network pruning, clustering and quantization schemes, as well as an end-to-end design space exploration flow for performance and efficiency optimization.

In the future, we plan to explore the possibility of applying mixed-precision quantification to existing architectures to further reduce the storage and bandwidth requirements of model deployments. In addition, to prove that our schemes are also applicable to other CNN-based object detection algorithms and accelerator designs, we will also try to apply proposed architecture to implement to two-stage-based detection frameworks, such as Faster-RCNN.

ACKNOWLEDGMENT

The authors would like to thank the anonymous reviewers for their valuable comments and constructive suggestions, which helped in improving the quality of the paper.

REFERENCES

[1] R. Girshick, "Fast R-CNN," in *Proc. IEEE Int. Conf. Comput. Vis. (ICCV)*, Dec. 2015, pp. 1440–1448.

[2] 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.

[3] W. Liu, D. Anguelov, D. Erhan, C. Szegedy, S. E. Reed, C. Fu, and A. C. Berg, "SSD: Single shot multibox detector," in *Proc. Eur. Conf. Comput. Vis. (ECCV)*, 2016, pp. 21–37.

[4] 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.

[5] J. Redmon and A. Farhadi, "YOLO9000: Better, faster, stronger," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit. (CVPR)*, Jul. 2017, pp. 6517–6525.

[6] P. Viola and M. J. Jones, "Robust real-time face detection," *Int. J. Comput. Vis.*, vol. 57, no. 2, pp. 137–154, May 2004.

[7] N. Dalal and B. Triggs, "Histograms of oriented gradients for human detection," in *Proc. IEEE Comput. Soc. Conf. Comput. Vis. Pattern Recognit. (CVPR)*, vol. 1, Jun. 2005, pp. 886–893.

[8] P. Felzenszwalb, D. McAllester, and D. Ramanan, "A discriminatively trained, multiscale, deformable part model," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit. (CVPR)*, Jun. 2008, pp. 1–8.

[9] S. Han, J. Pool, J. Tran, and W. J. Dally, "Learning both weights and connections for efficient neural networks," in *Proc. 28th Adv. Neural Inf. Process. Syst.*, 2015, pp. 1135–1143.

[10] P. Molchanov, S. Tyree, T. Karras, T. Aila, and J. Kautz, "Pruning convolutional neural networks for resource efficient inference," in *Proc. Int. Conf. Learn. Represent. (ICLR)*, 2017, pp. 1–17.

[11] H. Mao, S. Han, J. Pool, W. Li, X. Liu, Y. Wang, and W. J. Dally, "Exploring the regularity of sparse structure in convolutional neural networks," *CoRR*, vol. abs/1705.08922, 2017. [Online]. Available: http://arxiv.org/abs/1705.08922

[12] K. Ullrich, E. Meeds, and M. Welling, "Soft weight-sharing for neural network compression," in *Proc. Int. Conf. Learn. Represent. (ICLR)*, 2017, pp. 1–16.

[13] P. Gysel, J. Pimentel, M. Motamedi, and S. Ghiasi, "Ristretto: A framework for empirical study of resource-efficient inference in convolutional neural networks," *IEEE Trans. Neural Netw. Learn. Syst.*, vol. 29, no. 11, pp. 5784–5789, Nov. 2018.

[14] S. Zhou, Z. Ni, X. Zhou, H. Wen, Y. Wu, and Y. Zou, "Dorefa-Net: Training low bitwidth convolutional neural networks with low bitwidth gradients," *CoRR*, vol. abs/1606.06160, 2016. [Online]. Available: http://arxiv.org/abs/1606.06160

[15] M. Courbariaux, Y. Bengio, and J. David, "Binaryconnect: Training deep neural networks with binary weights during propagations," in *Proc. Adv. Neural Inf. Proces. Syst.*, 2015, pp. 3123–3131.

[16] D. Wang, J. An, and K. Xu, "PipeCNN: An OpenCL-based FPGA accelerator for large-scale convolution neuron networks," *CoRR*, vol. abs/1611.02450, 2016. [Online]. Available: http://arxiv.org/abs/1611.02450

[17] C. Ding, S. Wang, N. Liu, K. Xu, Y. Wang, and Y. Liang, "REQ-YOLO: A resource-aware, efficient quantization framework for object detection on FPGAs," in *Proc. ACM/SIGDA Int. Symp. Field-Program. Gate Arrays*, Feb. 2019, pp. 33–42.

[18] H. Nakahara, H. Yonekawa, T. Fujii, and S. Sato, "A lightweight YOLOv2: A binarized CNN with a parallel support vector regression for an FPGA," in *Proc. ACM/SIGDA Int. Symp. Field-Program. Gate Arrays*, Feb. 2018, pp. 31–40.

[19] N. P. Jouppi, C. Young, N. Patil, and D. Paerson, "A domain-specific architecture for deep neural networks," *Commun. ACM*, vol. 61, no. 9, pp. 50–59, 2018, doi: 10.1145/3154484.

[20] A. Monakov, A. Lokhmotov, and A. Avetisyan, "Automatically tuning sparse matrix-vector multiplication for GPU architectures," in *High Performance Embedded Architectures and Compilers* (Lecture Notes in Computer Science). Berlin, Germany: Springer-Verlag, 2010, pp. 111–125.

[21] W. A. Sufah and K. Ahmad, "On implementing sparse matrix multi-vector multiplication on GPUs," in *Proc. IEEE Int. Conf High Perform. Comput. Commun. 6th Int. Symp. Cyberspace Saf. Secur. 11th Int. Conf. Embedded Softw. Syst. (HPCC, CSS, ICESS)*, Aug. 2014, pp. 1117–1124.

[22] K. Xu, X. Wang, and D. Wang, "A scalable OpenCL-based FPGA accelerator for YOLOv2," in *Proc. IEEE 27th Annu. Int. Symp. Field-Program. Custom Comput. Mach. (FCCM)*, Apr. 2019, p. 317.

[23] H. Nakahara, M. Shimoda, and S. Sato, "A demonstration of FPGA-based you only look once version2 (YOLOv2)," in *Proc. 28th Int. Conf. Field Program. Log. Appl. (FPL)*, Aug. 2018, pp. 457–4571.

[24] 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, Autom. Test Eur. Conf. Exhib. (DATE)*, Mar. 2018, pp. 704–708.

[25] X. Xu and B. Liu, "FCLNN: A flexible framework for fast CNN prototyping on FPGA with OpenCL and caffe," in *Proc. Int. Conf. Field-Program. Technol. (FPT)*, Dec. 2018, pp. 238–241.

[26] J. Ma, L. Chen, and Z. Gao, "Hardware implementation and optimization of tiny-YOLO network," *Commun. Comput. Inf. Sci.*, vol. 815, no. 1, pp. 224–234, 2018.

[27] D. Wang, K. Xu, Q. Jia, and S. Ghiasi, "ABM-SpConv: A novel approach to FPGA-based acceleration of convolutional neural network inference," in *Proc. 56th Annu. Design Autom. Conf.*, Jun. 2019, pp. 1–6.

[28] S. Winograd, *Arithmetic Complexity of Computations*. Philadelphia, PA, USA: SIAM, 1980.

[29] A. Aimar, H. Mostafa, E. Calabrese, A. Rios-Navarro, I.-A. Lungu, R. Tapiador-Morales, M. B. Milde, F. Corradi, A. Linares-Barranco, S.-C. Liu, and T. Delbruck, "NullHop: A flexible convolutional neural network accelerator based on sparse representations of feature maps," *IEEE Trans. Neural Netw. Learn. Syst.*, vol. 30, no. 3, pp. 644–656, Mar. 2019.

[30] C. Zhu, K. Huang, S. Yang, Z. Zhu, H. Zhang, and H. Shen, "An efficient hardware accelerator for structured sparse convolutional neural networks on FPGAs," *CoRR*, vol. abs/2001.01955, 2020. [Online]. Available: http://arxiv.org/abs/2001.01955

[31] R. Girshick, J. Donahue, T. Darrell, and J. Malik, "Rich feature hierarchies for accurate object detection and semantic segmentation," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, Jun. 2014, pp. 580–587.

[32] *ShuangXieIrene/ssds.pytorch*. Accessed: Mar. 16, 2020. [Online]. Available: https://github.com/ShuangXieIrene/ssds.pytorch

[33] S. Han, H. Mao, and W. Dally, "Deep compression: Compressing deep neural networks with pruning, trained quantization and Huffman codings," in *Proc. Int. Conf. Learn. Represent. (ICLR)*, 2016, pp. 1–14.

[34] Intel Corporation. *Intel FPGA SDK for OpenCL Pro Edition Programming Guide V19.3*. Accessed: Feb. 14, 2020. [Online]. Available: https://www.intel.com/

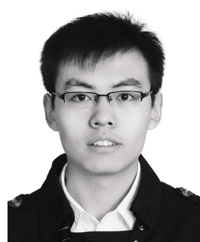
[35] U. Aydonat, S. O'Connell, D. Capalija, A. C. Ling, and G. R. Chiu, "An OpenCL(TM) deep learning accelerator on Arria 10," in *Proc. ACM/SIGDA Int. Symp. Field-Program. Gate Arrays*, 2017, pp. 55–64.

[36] 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.

[37] S. Williams, A. Waterman, and D. Patterson, "Roofline: An insightful visual performance model for multicore architectures," *Commun. ACM*, vol. 52, no. 4, pp. 65–76, 2009.

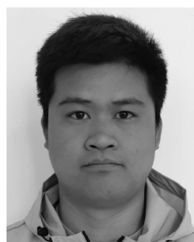
[38] T. Zhang, R. Ramakrishnan, and M. Livny, "BIRCH: An efficient data clustering method for very large databases," in *Proc. ACM SIGMOD Int. Conf. Manage. Data (SIGMOD)*, 1996, pp. 103–114.

[39] 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. (CVPR)*, Jun. 2018, pp. 2704–2713.



ZIXIAO WANG received the B.S. degree from Hebei Normal University, China, in 2018. He is currently pursuing the M.S. degree with the Institute of Information Science, Beijing Jiaotong University, Beijing, China. His research interests include heterogeneous computing systems, neural network compression, and high-performance computing architectures for deep learning applications.

KE XU received the B.S. degree from the Hefei University of Technology, China, in 2016. He is currently pursuing the Ph.D. degree with the Institute of Information Science, Beijing Jiaotong University, Beijing, China. His research interests include neural network compression, high-performance computing architectures for embedded applications, and computer vision.



SHUAIXIAO WU received the B.E. degree from Beijing Information Science and Technology University, China, in 2019. He is currently pursuing the M.E. degree in electronics and communication engineering with the School of Computer and Information Technology, Beijing Jiaotong University. His research interests include computer arithmetic for reconfigurable devices and high-performance and energy efficient computing architectures for embedded and machine learning applications.

LI LIU received the bachelor's degree in EE from Tsinghua University and the Ph.D. degree in CS from the University of Missouri in 2009. He joined Marvell Semiconductor and then Realtek, USA, as a Senior Algorithm Engineer, with a focus on video codec and image processing. He is currently with the Heterogeneous Computing Group, Kuaishou Technology, Palo Alto, CA, USA, as a Heterogeneous Platform Architect.

LINGZHI LIU (Senior Member, IEEE) received the B.S. degree from Xi'an Jiaotong University, Xi'an, China, in 1998, and the Ph.D. degree from Shanghai Jiaotong University, Shanghai, China, in 2004. He was the Senior Manager of Algorithm and Architecture Department, Realtek, USA, from 2014 to 2018. He was a Postdoctoral Researcher with EE Department, University of Washington, from 2005 to 2008. He is the Location Manager of the U.S. Research and Development Center, and the Head and the Chief Architect of the Heterogeneous Computing Group, Kuaishou Technology, Palo Alto, CA, USA. His general interests include neural network algorithm and architecture, multimedia algorithm and implementation, VLSI systems, ASIC, and FPGA design. He is a TC Member of the IEEE Visual Signal Processing and Communications.

DONG WANG (Member, IEEE) received the M.S. and Ph.D. degrees in electronic engineering from Xi'an Jiaotong University, Xi'an, China, in 2010 and 2006, respectively. He was a Visiting Scholar with the Department of Electrical and Computer Engineering, University of California at Davis, Davis, CA, USA, from 2018 to 2019. He is an Associate Professor with the Institute of Information Science, Beijing Jiaotong University, Beijing, China. His research interests include computer arithmetic for reconfigurable devices and high performance and energy efficient computing architectures for embedded and machine learning applications.