End-to-End FPGA-based Object Detection Using Pipelined CNN and Non-Maximum Suppression

Anupreetham Anupreetham[1], Mohamed Ibrahim[2,4], Mathew Hall[2], Andrew Boutros[2,3,4], Ajay Kuzhively[1], Abinash Mohanty[1], Eriko Nurvitadhi[4], Vaughn Betz[2,3], Yu Cao[1], Jae-sun Seo[1]
[1]*Arizona State University* [2]*University of Toronto* [3]*Vector Institute for AI* [4]*Intel Corporation*
E-mails: {anolas11, yu.cao, jseo28}@asu.edu, mohamed1.ibrahim@intel.com, vaughn@eecg.utoronto.ca

***Abstract*—Object detection is an important computer vision task, with many applications in autonomous driving, smart surveillance, robotics, and other domains. Single-shot detectors (SSD) coupled with a convolutional neural network (CNN) for feature extraction can efficiently detect, classify and localize various objects in an input image with very high accuracy. In such systems, the convolution layers extract features and predict the bounding box locations for the detected objects as well as their confidence scores. Then, a non-maximum suppression (NMS) algorithm eliminates partially overlapping boxes and selects the bounding box with the highest score per class. However, these two components are strictly sequential; a conventional NMS algorithm needs to wait for all box predictions to be produced before processing them. This prohibits any overlap between the execution of the convolutional layers and NMS, resulting in significant latency overhead and throughput degradation. In this paper, we present a novel NMS algorithm that alleviates this bottleneck and enables a fully-pipelined hardware implementation. We also implement an end-to-end system for low-latency SSD-MobileNet-V1 object detection, which combines a state-of-the-art deeply-pipelined CNN accelerator with a custom hardware implementation of our novel NMS algorithm. As a result of our new algorithm, the NMS module adds a minimal latency overhead of only 0.13μs to the SSD-MobileNet-V1 convolution layers. Our end-to-end object detection system implemented on an Intel Stratix 10 FPGA runs at a maximum operating frequency of 350 MHz, with a throughput of 609 frames-per-second and an end-to-end batch-1 latency of 2.4 ms. Our system achieves 1.5× higher throughput and 4.4× lower latency compared to the current state-of-the-art SSD-based object detection systems on FPGAs.**

I. Introduction

Recent advances in deep learning (DL) are continuously improving the quality of results achieved in many applications. This includes many computer vision tasks [1], natural language processing [2], etc. Object detection is a key computer vision task, with a myriad of use cases in various domains such as autonomous vehicles, traffic monitoring, manufacturing, robotics, image search, and smart surveillance [3]. Most of these use cases require not only high detection accuracy, but also low-latency real-time performance. For example, low latency is of crucial importance when detecting a pedestrian and applying the brakes in an autonomous vehicle.

Object detection systems typically consist of feature extraction followed by detection and localization of objects. Deep neural networks, and more specifically convolutional neural networks (CNNs), are recently being used as universal feature extractors that result in improved detection accuracy compared to hand-crafted feature extraction approaches [4]. However, CNNs are very compute intensive and therefore a light-weight model is favorable to reduce the overall computational complexity and meet the stringent real-time requirements. For example, single-shot detector (SSD) [5] coupled with MobileNet-V1 [6] as its feature extractor is a popular object detection pipeline which is aggressively optimized for real-time applications. This workload, SSD-MobileNet-V1, is one of the MLPerf inference workloads [7] for benchmarking and comparing DL hardware acceleration solutions.

The SSD component of the object detection pipeline consists of feed-forward convolution layers followed by a sequence of preprocessing operations and non-maximum suppression (NMS). The SSD convolution layers consume the features extracted by the CNN and use them to generate predictions of bounding boxes and scores for detected objects at different scales. These predictions go through a series of preprocessing operations and then an NMS module sorts all bounding boxes, eliminates partially overlapping boxes and selects the bounding box with the highest confidence score per class. This dataflow, which is used in all prior works, imposes a strictly sequential dependency between the convolutional layers and the NMS algorithm. The NMS component must wait for all bounding boxes to be produced before processing them, resulting in a substantial latency overhead for the whole system.

In this work, we devise a novel NMS algorithm that eliminates this strict sequential dependency and enables overlapping the execution of the convolutional layers with the NMS preprocessing and NMS components. This allows the implementation of a fully-pipelined streaming NMS hardware module, which significantly reduces latency. Then, we modify and integrate HPIPE, a state-of-the-art deeply pipelined CNN accelerator, with our new NMS module to implement an end-to-end object detection system on an Intel Stratix 10 FPGA. Our deployed system outperforms all existing FPGA-based object detection solutions, with a throughput of 609 frames-per-second (FPS) and an end-to-end latency of only 2.4 ms based on actual hardware measurements. In summary, our contributions are as follows:

- We present a novel NMS algorithm that eliminates the sequential dependency on the preceding convolutional layers.
- We implement a fully-pipelined streaming NMS module that exploits our novel NMS algorithm.
- We implement an end-to-end FPGA-based object detection system that achieves 1.5× higher throughput and 4.4× lower latency than the current state-of-the-art.

II. Background and Related Work

Object detection systems are typically composed of two main stages. Relevant features are first extracted from an input image, and then an object detector is used to classify and localize different types of objects. In this section, we will give a brief overview on these two components and prior works on FPGA-based object detection.

1946-1488/21/$31.00 ©2021 IEEE
DOI 10.1109/FPL53798.2021.00021

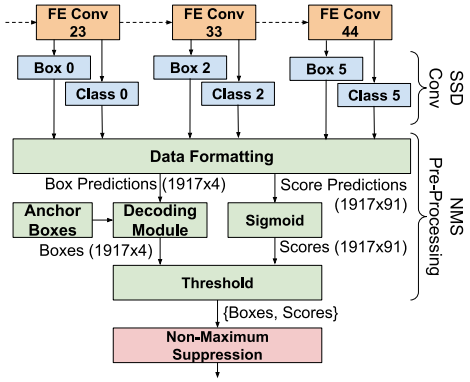

Fig. 1: SSD-MobileNet-V1 structure overview.

A. CNN-based Feature Extraction

Classically, domain experts had to devise hand-crafted feature extractors such as the histogram of oriented gradients [8] or the scale-invariant feature transform [9]. These feature extractors take an input image and produce a feature vector (i.e. an intermediate representation) that is then fed into a pre-trained classifier/detector. These features used to be very specific to a certain domain or application and required tedious tuning to adopt for another domain or dataset. In recent years, CNNs have been widely replacing hand-crafted techniques as universal feature extractors to achieve better quality of results at the cost of higher computational complexity. More recently, new classes of light-weight networks were introduced to reduce the computational complexity of CNNs and enable their use in real-time mobile applications. One class of these networks is MobileNets [6], which we focus on in this work. MobileNets reduce the computational complexity by replacing standard convolutions with depthwise separable convolutions, a combination of a depthwise convolution followed by a pointwise convolution [10]. This network structure significantly reduces the number of model parameters and operations. The specific model that we use in this work, MobileNet-V1 [6], has 4.2 million weight parameters and can fit in the on-chip memory of most modern FPGAs [11].

B. Single-Shot Detector and Non-Maximum Suppression

Object detectors can either be two-stage or one-stage detectors [12]. A two-stage detector first generates candidate object bounding boxes, then these boxes are classified in the second stage. In general, two-stage detectors can achieve high accuracy but they are computationally intensive. On the other hand, one-stage detectors eliminate the proposal generation stage and are generally faster than their two-stage counterparts. Examples of one-stage detectors are SSD [5] and YOLOv3 [13]. In this work, we focus on one-stage detectors, more specifically SSD, since it is the one used in the SSD-MobileNet-V1 model from the MLPerf workloads.

As shown in Fig. 1, the SSD component in SSD-MobileNet-V1 includes 6 box predictor and 6 class predictor convolution layers. These 12 layers get their inputs from different intermediate convolution layers in the feature extraction network. The 6 box predictor layers generate $1,917$ bounding box predictions specified as the relative coordinates of the box with respect to the *anchor boxes*. Anchor boxes are defined during model training and represent the ideal locations, shapes and sizes of bounding boxes for objects the model is aiming to detect. Convolution layers of SSD-MobileNet-V1 produce outputs with spatial dimensions of shape $(19 \times 19 \times 12)$, $(10 \times 10 \times 24)$, $(5 \times 5 \times 24)$, $(3 \times 3 \times 24)$, $(2 \times 2 \times 24)$ and $(1 \times 1 \times 24)$. For the first layer, each location of the 19×19 channels has 3 predefined boxes against which a box will be matched. Hence, this layer has $19 \times 19 \times 3 = 1083$ box predictions. Similarly the other 5 box prediction layers will have 6 predefined boxes per location, which results in 600 $(10 \times 10 \times 6)$, 150 $(5 \times 5 \times 6)$, 54 $(3 \times 3 \times 6)$, 24 $(2 \times 2 \times 6)$, 6 $(1 \times 1 \times 6)$ box predictions per layer. In total, all SSD layers combined will generate $1,917$ prediction boxes per image.

On the other hand, the 6 class predictor layers generate class prediction scores (out of 91 classes) for each of the $1,917$ generated boxes (each of which is described as `{x,y,w,h}`). These box and class predictions are produced from the convolution layers as 3D tensors in a format that is decided by the accelerator compute pattern. Then, these tensors go through a series of data formatting operations (e.g. permutations, reshapes and concatenations) to generate a list of $1,917$ boxes and scores for downstream processing. The list of generated boxes are then passed to a decoding module where they are matched against the predefined anchor boxes to generate the list of predicted boxes. Also, a sigmoid function is applied to the list of scores to generate the predicted scores for each of the 91 classes. The resulting lists are passed through a threshold function, such that only boxes with scores higher than a predefined threshold are processed by NMS.

The baseline NMS algorithm implemented in all prior SSD-based object detection systems is shown in Algorithm 1. For each of the 91 classes, the indexes of the boxes are sorted based on the descending order of the score of this class using the `argSort` function. As long as the list of sorted indexes is not empty, the box corresponding to the first score (i.e. current best candidate) is added as a selected box for this class. Then, the intersection over union (IOU) values between this box and all other boxes are calculated and compared to a predefined IOU threshold. Boxes with IOU values above this threshold are discarded by erasing them from the list of indexes. This process continues until all boxes are either selected or discarded for this class. Then, the selected boxes for this class are added to the list of all selected boxes before moving onto the next class.

The sorting operation at the beginning of the baseline NMS algorithm requires all boxes to be produced before starting the NMS computations. This requires a large amount of intermediate storage and imposes a strictly sequential dependency between the execution of the convolution layers and NMS computations, resulting in increased latency overhead and reduced throughput for the whole system. In addition, this algorithm and its preprocessing steps have a complex control flow that does not readily lend itself to an efficient hardware implementation (e.g. data format manipulations, erasing elements from a list) and are typically offloaded to the host CPU in most of prior works [14]. In this work, we present a novel NMS algorithm that eliminates this sequential dependency constraint and simplifies the control flow, enabling a fully-pipelined end-to-end hardware implementation on the FPGA with significantly reduced latency overhead.

C. FPGA-based Object Detection Systems

While many prior works have implemented object detection systems on FPGAs, a number of these works did not implement the NMS pre-processing and NMS modules on the FPGA, but rather offloaded them to the host CPU [14]–[16]. In contrast, our work

Algorithm 1: Baseline NMS algorithm

```
   Data: scores, boxes, IOU_thr
   Result: detected_objects
1  detected_objects = {};
2  for each class do
3      detected_class_objects = {};
4      indexes = argSort(scores, class);
5      while indexes not empty do
6          index = indexes[0];
7          best = {boxes[index], scores[index]};
8          detected_class_objects.append(best);
9          scores.erase(index); boxes.erase(index);
10         IOU_val = calculate_IOU(best, boxes);
11         indexes = filter(IOU_val, IOU_thr);
12     detected_objects.append(class_selected_boxes)
```

Algorithm 2: Novel NMS Implementation

```
   Data: scores, boxes, IOU_thr
   Result: detected_objects
1  Instantiate selected_boxes; //List of 65 empty boxes
2  for each box in boxes do
3      box_inserted = False;
4      for each sbox in selected_boxes do
5          if !box_inserted then
6              //If sbox is empty, insert new box to list
7              if sbox is empty then
8                  sbox = box;
9                  box_inserted = True;
10             //Else, replace it with new box if better
11             else
12                 box_inserted = replaceIf(box, sbox, IOU_thr);
13         //If new box is inserted, delete redundant sboxes
14         else
15             deleteIf(box, sbox, IOU_thr)
16 detected_objects = selected_boxes;

17 replaceIf (box, sbox, threshold) is
18     IOU = calculate_IOU(box, sbox);
19     if same_class(box, sbox) & IOU > threshold then
20         if box.score > sbox.score then
21             sbox.replaceWith(box);
22         return True;
23     return False;

24 deleteIf (box, sbox, threshold) is
25     IOU = calculate_IOU(box, sbox);
26     if same_class(box, sbox) & IOU > threshold &
         box.score > sbox.score then
27         delete sbox;
```

implements the end-to-end object detection system on the FPGA to achieve lower latency and higher throughput. The authors of [17] presented a YOLOv3 object detector with the DarkNet-53 model for feature extraction. They implemented a custom hardware module for NMS with bubble sorting that achieved a latency of 21 μs for $3,000$ bounding boxes. An FPGA-based architecture for SSDLite-MobileNet-V2 was also presented in [18]. The authors proposed a fused bottleneck residual block that significantly reduces the number of model parameters and overall latency. They achieved a throughput of 65 FPS with 20.3 mean average precision (mAP) on the COCO dataset for their design implemented on a Xilinx ZC706 FPGA. In [19], the authors implemented an MnasNet-based feature extractor along with a hardware NMS module on a Xilinx Virtex-7. They achieve a throughput of 23 FPS and 22.8 mAP on the COCO dataset. In contrast, our work focuses on the standard MLPerf object detection workload, SSD-MobileNet-V1. To the best of our knowledge, this work is the first to present a modified NMS algorithm that eliminates sequential dependencies between the convolutional layers and their post-processing stages.

III. Novel NMS Algorithm & SSD Hardware Implementation

A. Novel NMS Algorithm

Our novel NMS algorithm is shown in Algorithm 2. Unlike the baseline NMS in Algorithm 1, our algorithm does not differentiate between input prediction boxes from different classes and does not start by sorting them based on their prediction scores. Thus, this algorithm does not need to wait for all boxes to be ready before it starts processing. This allows a streaming hardware implementation that processes an input prediction box as soon as it is ready. Our algorithm keeps track of a list of selected boxes that starts empty for every image. For each input box prediction, it is first declared as a candidate to be inserted (line 3) and then we loop over the elements in the list of selected boxes to compare the input box prediction with all previously selected boxes. If the current entry in the selected boxes list is empty, the input box is inserted to the list (line 8) and the flag is set to true to indicate that it was already inserted (line 9). If the current selected boxes entry is not empty, it is compared and replaced by the input box if it is a better candidate (line 12, 17-23). After an input box is selected, it still needs to be compared to the rest of the selected boxes list to delete any previously selected box from the same class with a lower prediction score and higher-than-threshold IOU (lines 15, 24-27).

Although our algorithm does not sort the scores, it achieves the same functionality of the baseline algorithm. The sorting is performed implicitly as we compare and replace/remove selected boxes based on both their score and calculated IOU. In the baseline algorithm, since the boxes were initially sorted based on their scores, this selection only involves the calculated IOU.

B. NMS Preprocessing Hardware Implementation

The proposed architecture of the NMS preprocessing and NMS modules is presented in Fig. 2. The convolution layers of SSD predicts $1,917$ box coordinates as described in Section II, each with score predictions for the 91 classes. These outputs are processed in parallel by the NMS preprocessing module to consolidate the boxes for the final detection.

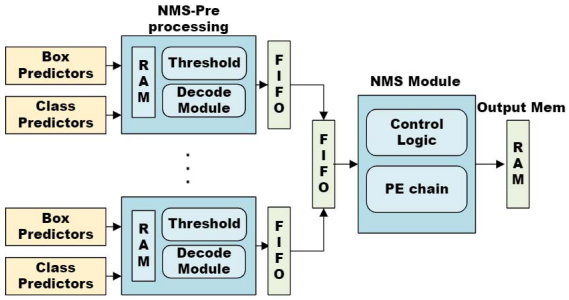

Fig. 2: Architecture of NMS preprocessing and NMS modules.

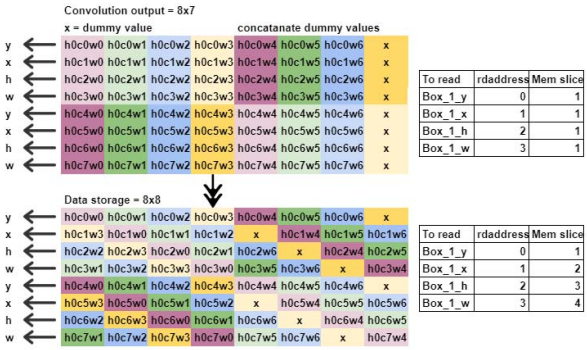

Fig. 3: Original vs. modified memory data arrangement to store anchor boxes and intermediate convolution results.

1) Data formatting: As will be detailed in Section IV, the state-of-the-art CNN accelerator that we use in this work, HPIPE, produces tensor outputs of the convolution layers in HCW format. This means that width dimension (W) is produced first, then channel dimension (C) followed by height dimension (H). We eliminate the need for any permute operations by storing the SSD convolution outputs on-the-fly in blocks of 4×4 circulant matrices which are circularly rotated at every row as shown in the lower half of Fig. 3, similar to [20]. To achieve this, we concatenate n columns of dummy values to the outputs of convolution layers such that $n = \lceil \text{column}/4 \rceil$. These dummy values are zeroes for the box prediction and the least possible negative number in the fixed point representation for the class prediction. The four columns of the 4×4 blocks are stored in different block RAMs (BRAMs), so that all of them can be read in parallel as shown in Fig. 3. In total, we need 104 M20K BRAMs to store both box predictions and class predictions from all the SSD convolution layers. Similarly, the pre-defined $1,917$ anchor boxes are first split into six groups for the six SSD box prediction convolution layers and pre-processed offline to match the same address locations as their corresponding box predictions.

2) Thresholding: Typically, thresholding is done at the end of the NMS pre-processing (Fig. 1). However, in our hardware implementation, we push the thresholding stage to directly sample and threshold the convolution output score predictions. This helps minimize the storage of intermediate data for NMS pre-procesing and reduces the total amount of computation. Only the corresponding boxes that exceed the threshold are further processed to generate the final coordinates of anchor boxes. To achieve that, we map the

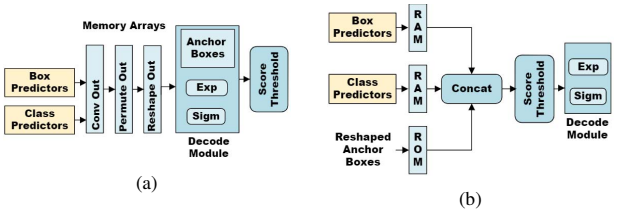

Fig. 4: The NMS preprocessing implementations for (a) the conventional scheme and (b) our optimized scheme.

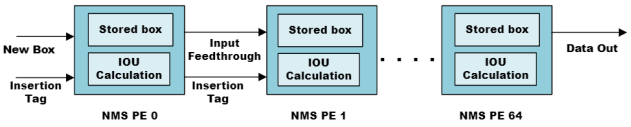

Fig. 5: NMS hardware implementation.

original threshold value using an inverse sigmoid function to obtain the correct threshold for our approach. For example, a threshold value of 0.3 at the output of the sigmoid function corresponds to a threshold value of -0.84 at the convolution output.

3) Decoding: The `{x,y,w,h}` values of the box predictions which pass the score threshold are coupled with their corresponding anchor values. These are then used to calculate the final `{x_min,y_min,x_max,y_max}` coordinates of the boxes that are given as input to the NMS module. These calculations involve a series of multiplication, constant division, and exponentiation operations. In our implementation, we convert the constant divisions to multiplications to reduce complexity and use vendor-specific IPs for multiplications, additions, and exponentiation. We have 6 different instances of the decoding module that work on the output of the 6 SSD box prediction convolution layers in parallel.

Fig. 4 compares the conventional NMS preprocessing architecture (Fig. 4a) to our optimized implementation (Fig. 4b). We avoid the need for multiple read/write operations by eliminating the explicit permute and reshape operations, and we do not process all the box predictions by pushing the threshold operation before the decode stage.

C. NMS Hardware Implementation

Fig. 5 illustrates the hardware implementation of our novel NMS approach. It consists of a chain of processing elements (PEs), each of which stores the coordinates and score of a single bounding box (i.e. an entry of the `selected_boxes` list in Algorithm 2). Each PE in the chain receives a new input box, uses it to perform a set of operations and comparisons in 18 clock cycles, before passing it to the next PE in a pipelined daisy-chain fashion. Each PE has its own control logic that decides to replace the locally stored box with the new input box, ignore the new input box, or just delete the stored box. Each box passing through the chain of PEs is accompanied by an *insertion tag*, which determines whether it was previously stored in one of the PEs or not. The NMS module does not need to wait for all the input boxes to be produced by the SSD convolution layers and NMS pre-processing. Instead, it uses a FIFO to buffer the inputs, as shown in Fig. 2, which will be processed in a streaming fashion when the NMS logic is free.

After completely processing all the boxes of an image, the top-level module starts draining the PE chain to extract the locally stored boxes. However, not all the PEs in the chain are guaranteed

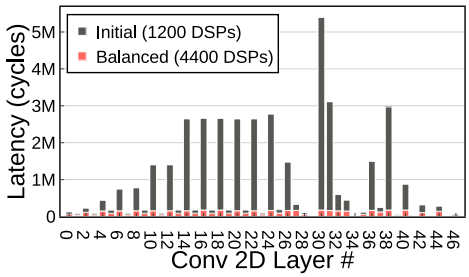

Fig. 6: Throughput balancing between layers in HPIPE.

to have valid stored boxes (i.e. some boxes can be deleted in the middle of the PE chain). Therefore, a simple output read logic is implemented to extract only the valid boxes and convert them from the `{x_min,y_min,x_max,y_max}` format to the final object detection outputs in the `{x,y,w,h}` format.

IV. End-to-End Object Detection System

We implement an end-to-end object detection system which consists of two major components: (1) the CNN accelerator for the feature extraction and SSD convolution layers, and (2) the SSD implementation we described in Section III. For the CNN accelerator, we use an extended version of HPIPE [21], a state-of-the-art deeply-pipelined inference accelerator for FPGAs.

A. HPIPE: Deeply-Pipelined CNN Accelerator

1) Architecture Details: HPIPE is a sparsity-aware deeply pipelined architecture that statically splits device resources across different CNN layers and builds customized hardware for every layer. Customized per-layer hardware enables high efficiency, while deep, interconnect-aware pipelining leads to a high operating frequency. HPIPE supports various layer types including standard, depthwise and pointwise convolution, sparse convolution that skips zero weights, max-pooling, rectified linear units (ReLU) and sigmoid activations. The layers are connected to each other through a latency-insensitive FIFO interface, and coarse back-pressure signals between the layers ensure functionality correctness. While HPIPE achieves high performance, processing all layers in a pipeline increases the bandwidth needed for weights, so it best suits networks where all weights can fit in the on-chip memory of FPGAs. The SSD-MobileNet-V1 network that we study in this paper has 4.2 million parameters. With all weights using 16-bit fixed-point precision, the whole model can easily fit in our target FPGA (Intel Stratix 10 GX2800) that has 244 Mb of on-chip memory storage (BRAMs). We choose HPIPE as the CNN accelerator for our feature extraction and SSD convolution layers as it achieves significantly higher performance than other FPGA-based accelerators [21].

2) Software Flow: The HPIPE tool flow takes a TensorFlow graph describing a CNN, along with a parameter file that specifies resource limits on the target FPGA as inputs. The compiler first performs some optimizations on the given TensorFlow graph in which some nodes are merged together and mapped to layers that HPIPE has a hardware implementation for. The compiler then starts the resource allocation and throughput balancing process, in which different layers of the CNNs will be allocated resources, mainly DSP blocks, based on their expected cycle latency. Fig. 6 shows an example of the throughput balancing process for the SSD-MobileNet-V1 convolution layers. The minimum parallelism settings are initially

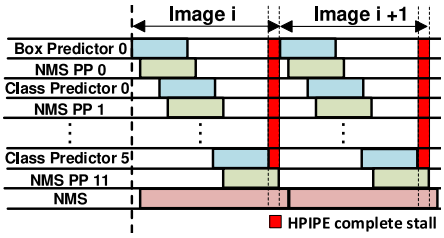

Fig. 7: Example execution trace of the end-to-end system.

used for all the layers in the model, resulting in large variation in the latencies of different layers as indicated by the grey bars in Fig. 6. Since the throughput of the pipeline is determined by its slowest stage, the compiler aims to balance layer latencies by increasing the parallelism factor and assigning more resources to the layer with the lowest throughput (i.e. highest number of cycles) until it is no longer the pipeline bottleneck. The process then repeats with other layers until the target resource utilization is met. The red bars in Fig. 6 show the final balanced latencies of the SSD-MobileNet-V1 layers at a target utilization of $4,400$ DSPs.

B. System Integration

Several modifications to the baseline HPIPE implementation were necessary to integrate it with our pipelined SSD hardware implementation from Section III. First, we added support for the additional SSD convolution layers including the six class-predictor and six box-predictor layers. Second, we modified the HPIPE interface such that the output of all the 12 SSD convolution layers are supplied to the NMS module as shown in Fig. 1. The output box and class predictions are then stored in RAMs (as shown in Fig. 2) to be later consumed by the NMS pre-processing module. Conventional SSD post-processing would require all the outputs of all SSD convolutions to be ready before starting execution, creating long pipeline stalls that significantly degrade HPIPE's throughput. This motivates our novel NMS algorithm and its streaming pipelined implementation which relaxes these constraints and eliminates the majority of pipeline stalls. Our implementation only needs the first four outputs of the SSD convolutions (`{x,y,w,h}`), after which the generation of convolution and SSD outputs are pipelined.

Fig. 7 shows an example execution trace from our system that shows the execution duration of SSD convolution layers (blue), NMS pre-processing (green), and NMS (pink). It highlights the pipelined execution of the SSD convolution layers in HPIPE and how they overlap with the execution of our streaming hardware implementation of the NMS pre-processing and NMS stages. The red vertical strips highlight the small portion of time in which an SSD-related overhead is not hidden by HPIPE computations (i.e. HPIPE is completely stalled waiting for the NMS pre-processing to finish execution). This portion is very small; about only 3% of the overhead added by the SSD component of object detection.

TABLE I: Breakdown of FPGA resource utilization (NMS-PP: NMS pre-processing module).

	ALMs	DSPs	M20K BRAMs
Full System	575,394 (62%)	5,129 (89%)	7,659 (65%)
HPIPE	468,203 (50%)	4,434 (77%)	7,179 (61%)
NMS-PP	40,866 (4%)	265 (5%)	425 (4%)
NMS	45,838 (5%)	430 (7%)	0 (0%)

TABLE II: Comparison of our work to prior FPGA-based object detection systems on the COCO dataset.

	Fan et al. [18]	Zhao et al. [19]	Mobilint [7]	Cai et al. [14]	This Work
Feature Extraction	MobileNet-V2 [22]	MnasNet [23]	MobileNet-V1 [6]	MobileNet-V1 [6]	MobileNet-V1 [6]
Object Detector	SSDLite	custom SSD	SSD	SSD	SSD
# Parameters	2.79M	3.9M	4.2M	4.2M	4.2M
FPGA Device	Zynq ZC706	Virtex-7	Alveo U250	Arria 10	Stratix 10 GX2800
Process Technology	28nm	28nm	16nm	20nm	14nm
Power (W)	9.9	-	-	-	55
Energy Eff. (J/image)	0.15	0.73	-	-	0.09
Frequency (MHz)	100	-	250	-	350
Throughput (FPS)	65	23	410	108	609
Latency (ms)	15.43	-	10.64	-	2.4
mAP	20.3	22.8	23.028	16.8	22.5

V. Results

A. Experimental Setup

We implement and deploy our system on a Terasic DE10-Pro board [24] with the largest monolithic Intel Stratix 10 GX2800 FPGA, attached as a PCIe accelerator card to an Intel Xeon E5-2650 server with 12 double-threaded cores and 94 GB of RAM. In this setup, the host CPU sends input images to the FPGA accelerator and receives back output predictions over the PCIe link. We use Intel Quartus Prime Pro 19.4 to perform synthesis, place and route, Synopsys VCS for RTL-level simulations, and the Quartus power analyzer tool to obtain a vectorless power estimation for the system core. We verify our system's functionality on $4,800$ validation images from the COCO dataset. All the performance results reported in this section are based on real hardware measurements of our deployed system. We time the entire CPU host code starting from sending the first input image until receiving back the output predictions of the last image. Then, we use that to calculate the system performance results which includes all PCIe transfer times.

B. Implementation Results

Table I shows the resource utilization breakdown of different components of our end-to-end system. The results show that our system is DSP-bound with close to 90% utilization of the available DSP blocks. This highlights that scaling to a multi-FPGA solution with more DSPs available would allow HPIPE to assign more parallelism to different layers, further reducing the latency of the feature extraction and SSD convolution layers. However, we leave this out for future work. The results also show that both the NMS preprocessing and NMS modules are very lightweight; they consume less than 5% and 7% of the available FPGA resources, respectively.

C. Performance Results

We run our system at the maximum operating frequency reported by the Quartus timing analyzer, which is reported to be **350 MHz**. This results in a throughput of **609 FPS** with an end-to-end latency of **2.4 ms**. We measure a score of 22.5 mAP on the COCO validation dataset, which is higher than the MLPerf threshold (22 mAP). Power consumption is estimated to be **55 W**, which translates to an energy efficiency of **0.09 J/image**. In our hardware experiments, we found that we can over-clock the system up to a frequency of **460 MHz**, while maintaining correct functionality. This highlights the large timing safety margin added by the Quartus timing analyzer and matches with the observations from [25]. With over-clocking, our system can achieve a throughput of **812 FPS** and an end-to-end latency of **1.7 ms**, with the same mAP score. However, we do not use the over-clocking results when comparing to other prior works in the next subsection for the sake of having a fair comparison.

D. Comparison to Prior Works

Table II compares our work against other SSD-based object detection accelerators on FPGAs. The comparison shows that our accelerator achieves lower latency and higher throughput than all prior works, while achieving almost the same accuracy on the COCO dataset. Some of the prior works in the table are implemented on FPGAs from older processes; however, we still include them for completeness rather than for direct performance comparison. The most relevant comparison is against Mobilint [7], which is implemented on a same generation Xilinx FPGA, and is the only FPGA-based object detection submission for the MLPerf edge closed divison [26]. There is no publicly disclosed information about their implementation details, but comparing to their MLPerf submission results, our system achieves $4.4\times$ lower latency and $1.5\times$ higher throughput, while being only 0.5 of a point less mAP.

Fan et al. [18] implemented the SSDLite-MobileNet-V2 model which has lower computational complexity and fewer parameters compared to the SSD-MobileNet-V1 we implement. Despite that, our solution still achieves better throughput and latency at a higher detection accuracy. Their achieved throughput is the reciprocal of their end-to-end latency, which implies that there is no parallelism exploited between processing different images. In contrast, a key property of our system is the use of deeply pipelined CNN accelerator and SSD hardware implementation to achieve higher throughput.

VI. Conclusion

In this work, we implemented an end-to-end streaming accelerator for the standard MLPerf object detection workload, SSD-MobileNet-V1. We first devise a novel NMS algorithm that eliminates strict sequential dependency between convolution layers and NMS execution, then we present an optimized streaming hardware implementation for it. We also use and extend HPIPE, a state-of-the-art deeply-pipelined CNN accelerator, for processing the feature extraction and SSD convolutions. The combination of these 3 components results in a fully-pipelined end-to-end object detection system that achieves a throughput of 609 FPS and an end-to-end latency of 2.4 ms. This represents $1.5\times$ higher throughput and $4.4\times$ lower latency compared to the state-of-the-art SSD-based object detection system on FPGAs.

Acknowledgement

This work is partially supported by NSF grant 1652866, the Intel ISRA program on FPGA, JUMP C-BRIC (a SRC program sponsored by DARPA), the Intel/NSERC Industrial Research Chair in Programmable Silicon, and the Vector Institute for Artificial Intelligence. Any opinions, findings, conclusions or recommendations are those of the authors and not of the funding institutions.

REFERENCES

[1] A. Voulodimos *et al.*, "Deep Learning for Computer Vision: A Brief Review," *Computational Intelligence and Neuroscience*, 2018.

[2] D. W. Otter *et al.*, "A Survey of the Usages of Deep learning for Natural Language Processing," *Transactions on Neural Networks and Learning Systems*, 2020.

[3] Z. Zou *et al.*, "Object Detection in 20 Years: A Survey," *arXiv preprint arXiv:1905.05055*, 2019.

[4] A. Suleiman *et al.*, "Towards Closing the Energy Gap between HOG and CNN Features for Embedded Vision," in *International Symposium on Circuits and Systems (ISCAS)*, 2017.

[5] W. Liu *et al.*, "SSD: Single Shot Multibox Detector," in *European Conference on Computer Vision (ECCV)*, 2016.

[6] A. G. Howard *et al.*, "MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications," *arXiv preprint arXiv:1704.04861*, 2017.

[7] V. J. Reddi *et al.*, "MLPerf Inference Benchmark," in *International Symposium on Computer Architecture (ISCA)*, 2020.

[8] N. Dalal and B. Triggs, "Histograms of Oriented Gradients for Human Detection," in *Conference on Computer Vision and Pattern Recognition (CVPR)*, 2005.

[9] D. G. Lowe, "Object Recognition from Local Scale-Invariant Features," in *International Conference on Computer Vision (ICCV)*, 1999.

[10] F. Chollet, "Xception: Deep Learning with Depthwise Separable Convolutions," in *Conference on Computer Vision and Pattern Recognition (CVPR)*, 2017.

[11] A. Boutros and V. Betz, "FPGA Architecture: Principles and Progression," *IEEE Circuits and Systems Magazine*, vol. 21, no. 2, pp. 4–29, 2021.

[12] L. Jiao *et al.*, "A Survey of Deep Learning-based Object Detection," *IEEE Access*, vol. 7, pp. 128 837–128 868, 2019.

[13] J. Redmon and A. Farhadi, "YOLOv3: An Incremental Improvement," *arXiv preprint arXiv:1804.02767*, 2018.

[14] L. Cai *et al.*, "An FPGA Based Heterogeneous Accelerator for Single Shot MultiBox Detector (SSD)," in *International Conference on Solid-State & Integrated Circuit Technology (ICSICT)*, 2020.

[15] Y. Ma *et al.*, "Algorithm-hardware Co-design of Single Shot Detector for Fast Object Detection on FPGAs," in *International Conference on Computer-Aided Design (ICCAD)*, 2018.

[16] Z. Wang *et al.*, "Sparse-YOLO: Hardware/Software Co-design of an FPGA Accelerator for YOLOv2," *IEEE Access*, vol. 8, pp. 116 569–116 585, 2020.

[17] H. Zhang *et al.*, "Efficient Hardware Post Processing of Anchor-Based Object Detection on FPGA," in *International Symposium on VLSI (ISVLSI)*, 2020.

[18] H. Fan *et al.*, "A Real-Time Object Detection Accelerator with Compressed SSDLite on FPGA," in *International Conference on Field-Programmable Technology (FPT)*, 2018.

[19] T. Zhao *et al.*, "A Hardware Accelerator Based on Neural Network for Object Detection," in *Journal of Physics: Conference Series*, 2020.

[20] S. K. Venkataramanaiah *et al.*, "Automatic Compiler Based FPGA Accelerator for CNN Training," in *International Conference on Field Programmable Logic and Applications (FPL)*, 2019.

[21] M. Hall and V. Betz, "From TensorFlow Graphs to LUTs and Wires: Automated Sparse and Physically Aware CNN Hardware Generation," in *International Conference on Field Programmable Technology (FPT)*, 2020.

[22] M. Sandler *et al.*, "MobilenetV2: Inverted Residuals and Linear Bottlenecks," in *Conference on Computer Vision and Pattern Recognition (CVPR)*, 2018.

[23] M. Tan *et al.*, "MnasNet: Platform-Aware Neural Architecture Search for Mobile," in *Conference on Computer Vision and Pattern Recognition (CVPR)*, 2019.

[24] Terasic Inc., "DE10-Pro User Manual (GH1E1) v1.8," 2019.

[25] A. Boutros *et al.*, "Neighbors From Hell: Voltage Attacks Against Deep Learning Accelerators on Multi-Tenant FPGAs," *arXiv preprint arXiv:2012.07242*, 2020.

[26] MLPerf. MLPerf Inference v0.7 Results. https://mlperf.org/inference-results/ (Accessed on 31-03-2021).