High Throughput FPGA-Based Object Detection via Algorithm-Hardware Co-Design

ANUPREETHAM ANUPREETHAM, Arizona State University, USA
MOHAMED IBRAHIM, University of Toronto, Intel Corporation, Canada
MATHEW HALL, University of Toronto, Canada
ANDREW BOUTROS, University of Toronto, Vector Institute for AI, Canada
AJAY KUZHIVELY and ABINASH MOHANTY, Arizona State University, USA
ERIKO NURVITADHI, Intel Corporation, USA
VAUGHN BETZ, University of Toronto, Vector Institute for AI, Canada
YU CAO and JAE-SUN SEO, Arizona State University, USA

Object detection and classification is a key task in many computer vision applications such as smart surveillance and autonomous vehicles. Recent advances in deep learning have significantly improved the quality of results achieved by these systems, making them more accurate and reliable in complex environments. Modern object detection systems make use of lightweight convolutional neural networks (CNNs) for feature extraction, coupled with single-shot multi-box detectors (SSDs) that generate bounding boxes around the identified objects along with their classification confidence scores. Subsequently, a non-maximum suppression (NMS) module removes any redundant detection boxes from the final output. Typical NMS algorithms must wait for all box predictions to be generated by the SSD-based feature extractor before processing them. This sequential dependency between box predictions and NMS results in a significant latency overhead and degrades the overall system throughput, even if a high-performance CNN accelerator is used for the SSD feature extraction component. In this paper, we present a novel pipelined NMS algorithm that eliminates this sequential dependency and associated NMS latency overhead. We then use our novel NMS algorithm to implement an end-to-end fully pipelined FPGA system for low-latency SSD-MobileNet-V1 object detection. Our system, implemented on an Intel Stratix 10 FPGA, runs at 400 MHz and achieves a throughput of 2,167 frames per second with an end-to-end batch-1 latency of 2.13 ms. Our system achieves 5.3× higher throughput and 5× lower latency compared to the best prior FPGA-based solution with comparable accuracy.

CCS Concepts: • **Computer systems organization → Neural networks**; **Reconfigurable computing;**

This work is partially supported by NSF grant 1652866, the Intel ISRA program on FPGA, JUMP2.0 CoCoSys (a SRC program sponsored by DARPA), the Intel/VMware Crossroads 3D-FPGA Academic Research Center, 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.
Authors' addresses: A. Anupreetham, A. Kuzhively, A. Mohanty, Y. Cao, and J. Seo, Arizona State University, 781 S Terrace Rd, Tempe, AZ, USA, 85287; e-mails: {anolas11, akuzhive, amohant4, yu.cao, jaesun.seo}@asu.edu; M. Ibrahim, M. Hall, A. Boutros, and V. Betz, University of Toronto, 10 King's College Road Toronto, ON, Canada, M5S 3G4; e-mails: {mohamedabdelfattah.ibrahim, mathew.hall, andrew.boutros}@mail.utoronto.ca, vaughn@ece.utoronto.ca; E. Nurvitadhi, Intel Corporation, 2111 NE 25th Ave, Hillsboro, OR, USA, 97124; e-mail: eriko.nurvitadhi@gmail.com.

1936-7406/2024/01-ART1 $15.00
https://doi.org/10.1145/3634919

Additional Key Words and Phrases: FPGA accelerator, object detection, algorithm-hardware co-design, neural networks

ACM Reference format:
Anupreetham Anupreetham, Mohamed Ibrahim, Mathew Hall, Andrew Boutros, Ajay Kuzhively, Abinash Mohanty, Eriko Nurvitadhi, Vaughn Betz, Yu Cao, and Jae-sun Seo. 2024. High Throughput FPGA-Based Object Detection via Algorithm-Hardware Co-Design. *ACM Trans. Reconfig. Technol. Syst.* 17, 1, Article 1 (January 2024), 20 pages.
https://doi.org/10.1145/3634919

1 INTRODUCTION

Object detection involves identifying multiple objects in an input image or video frame, classifying them into different classes, and generating a bounding box with the detection confidence score around each object. Modern computer vision systems use object detectors powered by deep learning algorithms in various applications such as autonomous vehicles, traffic monitoring, manufacturing, robotics, image search, and smart surveillance [36]. Most of these applications 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 or an obstacle and applying the brakes in an autonomous vehicle. As such, an efficient implementation of high-performance low-latency object detection is necessary for the deployment of such computer vision systems.

Object detection consists of feature extraction followed by localization and classification of objects, as illustrated in Figure 1. Deep neural networks, and more specifically **convolutional neural networks (CNNs)**, have recently been used as universal feature extractors that result in improved detection accuracy, compared to classical hand-crafted feature extraction approaches [30]. After that, a **single-shot detector (SSD)** has been adopted in many modern systems for detection and localization of objects [19]. An SSD typically consists of a number of convolution layers followed by a sequence of data pre-processing operations and **non-maximum suppression (NMS)**. The SSD convolution layers use the features extracted by the CNN to generate predictions of bounding boxes and scores for the detected objects at different scales. These predictions go through a series of pre-processing operations. Then, the NMS module sorts all bounding boxes, eliminates partially overlapping boxes and selects the bounding box with the highest confidence score per class. To reduce the overall computational complexity, compact lightweight CNNs (e.g., MobileNet-V1 [13]) are commonly used for low-latency feature extraction. Hence, an SSD coupled with a MobileNet-V1 as its feature extractor, commonly known as SSD-MobileNet-V1, is a popular object detection pipeline which is aggressively optimized for real-time applications. This workload is selected as one of the MLPerf key inference workloads [23] for benchmarking deep learning hardware accelerators.

The dataflow of this workload, as well as any other SSD-based object detection, imposes a strictly sequential dependency between the convolution layers and the NMS module. The NMS component must wait for all bounding boxes to be produced before processing them, resulting in a substantial latency overhead and throughput degradation for the overall system. To address this, our recent work in [2] introduces a novel NMS algorithm that eliminates the strict sequential dependency and enables pipelining the execution of the convolution layers with the NMS components. We implement an end-to-end FPGA-based object detection system that uses our novel NMS algorithm to reduce the latency overhead and improve system throughput compared to traditional sequential implementations, with no effect on detection accuracy.

In this work, we further improve our NMS hardware implementation by interleaving the processing of multiple images within the NMS module (i.e., multi-threading). In addition, we improve

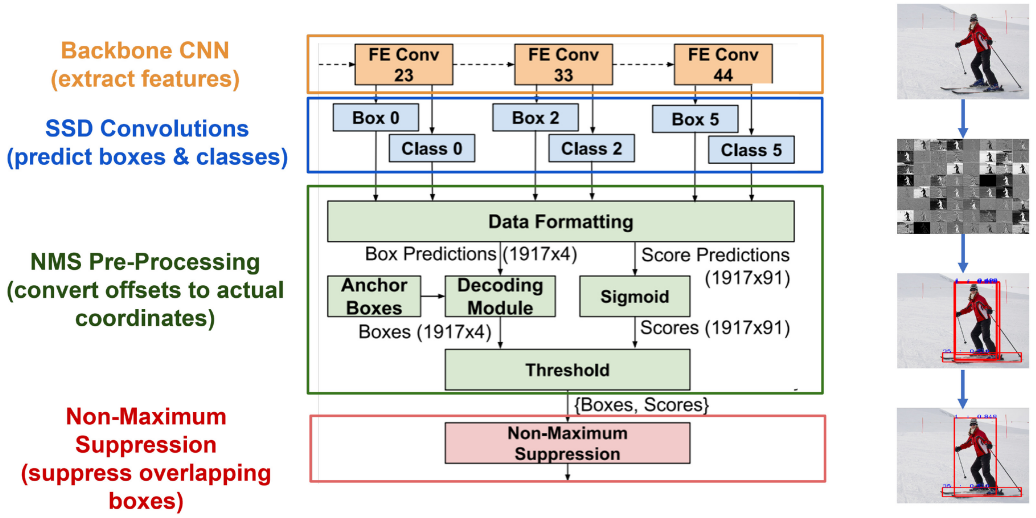


Fig. 1. Overview of the processing in SSD-based object detection.

the end-to-end object detection latency by implementing a piece-wise linear approximation for non-linear functions (e.g., sigmoid, exponential) instead of the more expensive floating-point-based implementation. Our end-to-end FPGA-based object detection system integrates HPIPE [11], a state-of-the-art deeply pipelined CNN accelerator, with our new fully-pipelined and multi-threaded NMS module. Our solution outperforms all existing FPGA-based object detection systems, with a throughput of 2,167 **frames per second (FPS)** and an end-to-end latency of only 2.13 ms based on actual hardware measurements on an Intel Stratix 10 FPGA.

Our key contributions can be summarized as follows:

- We present a novel pipelined NMS algorithm that eliminates the sequential dependency on the preceding convolution layers in SSD-based object detectors.
- We implement a fully-pipelined multi-threaded NMS hardware module that exploits our new NMS algorithm and interleaves the processing of multiple images to match the throughput of the layer-pipelined backbone CNN accelerator (HPIPE).
- We implement an end-to-end FPGA object detection system that outperforms prior FPGA systems with comparable accuracy and achieves 5.3× higher throughput and 5× lower latency compared to the only FPGA-based submission in 2021 MLPerf [23] for SSD-based object detection systems.

2 BACKGROUND

2.1 Object Detectors

Object detection is a fundamental task in many computer vision applications which involves the localization and classification of multiple objects within an image. The outputs of an object detection system are predictions of the coordinates of bounding boxes around the detected objects, as well as confidence scores for the classification of each of the localized objects (bounding boxes) into a number of pre-determined classes. Traditionally, two-stage detectors [15] were commonly used. The first stage of these detectors, known as the region proposal network, performs object localization and is then followed by a classifier to identify the class of the localized object. R-CNN [10] for instance, is a two stage detector which first generates bounding boxes based on a segmentation map of the image, and then a conventional CNN performs inference on each

of the generated boxes. Since CNN classification is applied on each of the generated *regions of proposal* (i.e., bounding boxes), this type of detector has a high computational complexity. A highly-accurate and optimized version of a VGG-16 R-CNN two-stage detector was presented in [26]. However, it could process only 5 images per second on an NVIDIA K40 28nm GPU, highlighting that two-stage object detectors are unsuitable for real-time applications.

More recently, one-stage detectors were introduced as a solution for the high computational complexity of two-stage detectors. This type of detector performs both localization and classification in a single forward pass of the model. The improvements of one-stage detectors over two-stage detectors arise from the use of a fixed set of pre-calculated boxes of varying dimensions and positions in the feature map. These boxes are known as *priors* or *anchor boxes* and are predefined based on the object sizes in the training dataset. One-stage detectors such as SSD [19], YOLO [24], and YOLOv3 [25] use anchor boxes to predict the bounding box of the detected objects in an input image. In our work, we focus on one-stage detectors, more specifically SSDs, since it is a popular choice for object detection pipelines and is used in the SSD-MobileNet-V1 model from the MLPerf benchmark suite. A final post-processing step known as non-maximum suppression (or NMS for short) is commonly used in YOLO/SSD-based object detectors to remove redundant bounding boxes around a localized and classified object.

2.2 CNN-Based Feature Extraction

CNNs have been widely used to perform image classification with high accuracy on complex datasets [12]. CNNs for image classification are typically composed of several convolution layers for feature extraction followed by a few fully-connected layers for classification. As they result in higher accuracy compared to classical hand-crafted feature extractors, SSDs commonly use the feature extraction layers of pre-trained image classification CNNs. Recently, a new class of CNN-based feature extractors called MobileNets [13] have become very popular due to their reduced computational cost compared to prior CNNs. This is achieved by replacing the traditional convolution operations by depthwise convolutions followed by pointwise convolutions [7] (collectively called *depthwise separable convolutions*). This reduces the total number of model parameters and operations with minimal impact on accuracy. Therefore, MobileNets have become favorable candidates for real-time applications, especially on mobile platforms with limited compute capabilities and power budgets. In this work, we use MobileNet-V1 [13] which has 4.2 million parameters and can fit in the on-chip memory of most modern FPGAs [3, 21].

2.3 Single-Shot Detectors

In SSD models, the feature extractor is followed by a series of SSD convolution layers that include box prediction and class prediction layers. As shown in Figure 1, the SSD convolutions are performed on intermediate feature maps of various scales, which allows the SSD model to detect objects of different sizes in an image. Feature maps produced by intermediate layers of the backbone feature extractor CNN are processed by pairs of SSD convolutional layers to predict both the coordinates of bounding boxes relative to the predefined priors (anchor boxes) as well as the classes of the objects inside these boxes. In this work, we use the COCO dataset [18] with 91 object classes, which is used in the MLPerf benchmarks.

The convolution layers of SSD-MobileNet-V1 produce outputs with (width $\times$ height $\times$ output channel) dimensions of ($19 \times 19 \times 12$), ($10 \times 10 \times 24$), ($5 \times 5 \times 24$), ($3 \times 3 \times 24$), ($2 \times 2 \times 24$) and ($1 \times 1 \times 24$). For the first layer, each location of the 19×19 channels has 3 pre-defined boxes against which a box will be matched. Hence, this layer has $19 \times 19 \times 3 = 1,083$ box predictions. Similarly, the other 5 box prediction layers will have 6 pre-defined 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 ($2 \times 2 \times 6$), and 6 ($1 \times 1 \times 6$) box predictions

ALGORITHM 1: Baseline NMS algorithm

```
Data: scores, boxes, IOU_thr
Result: detected_objects
detected_objects = {};
for each class do
    detected_class_objects = {};
    indexes = argSort(scores, class);
    while indexes not empty do
        index = indexes[0];
        best = {boxes[index], scores[index]};
        detected_class_objects.append(best);
        scores.erase(index); boxes.erase(index);
        IOU_val = calculate_IOU(best, boxes);
        indexes = filter(IOU_val, IOU_thr);
    detected_objects.append(class_selected_boxes)
```

per layer. In total, the combined SSD layers will generate 1, 917 prediction boxes per image. These box predictions are calculated as an offset from the stored priors or anchor boxes. On the other hand, the six class predictor layers generate class prediction scores (out of 91 classes) for each of the 1, 917 generated boxes, each of which is described with their center co-ordinates, width, and height as {cx,cy,w,h}.

Since the co-ordinates of the valid predicted boxes are currently in the form of an offset, they are compared against the stored anchor boxes and then are converted to actual co-ordinates in the NMS pre-processing step. The center and the size of the predicted boxes {cx,cy,w,h} are used to generate boundary co-ordinates $\{x_{min}, y_{min}, x_{max}, y_{max}\}$. Additionally, a sigmoid activation function is applied to the list of 91 class scores for each of the 1, 917 boxes. However, many of these predictions are then filtered out by comparing their confidence score against a pre-determined threshold. Only the boxes that pass this threshold are considered as valid boxes. Subsequently, the generated co-ordinates of the valid boxes along with their confidence scores are sent as candidates for further processing by the NMS stage.

2.4 Non-Maximum Suppression (NMS)

The baseline algorithm for NMS is shown in Algorithm 1. For every class in the dataset, the indexes of the boxes are sorted based on the descending order of the score using the argSort function. While the list of the sorted indexes is not empty, the confidence score of the box corresponding to the first index in the sorted list (i.e., the current best candidate) is added as a selected box for this class. Then, the **intersection over union (IOU)** ratio is calculated, as shown in Equation (1), between the current best candidate and all remaining boxes in the list.

$$Intersection\ Over\ Union\ (IOU) = \frac{Area\ of\ Overlap}{Area\ of\ Union} \tag{1}$$

If the IOU exceeds a pre-determined threshold (i.e., high overlapping area between two boxes), then both boxes likely contain the same object. Hence, the boxes with IOU values above the pre-determined threshold are discarded by deleting them from the list of candidates. This process continues until all boxes are either selected or discarded for a specific class. Then, the selected boxes for this class are added to the list of all selected boxes before proceeding to the next class.

The sorting operation at the beginning of baseline NMS requires all candidate boxes to be ready for the storing to be performed. 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 and reduced throughput for the overall system. In addition, this algorithm and its pre-processing 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 they are typically offloaded to the host CPU in prior works [6]. In our previous work [2], we introduced a novel implementation of the NMS algorithm that eliminates the sequential dependency, reduces intermediate storage requirement and simplifies the control flow, enabling a pipelined end-to-end hardware implementation on the FPGA with significantly reduced latency overhead. Despite this highly optimized unit, NMS remained as the bottleneck in the object detection pipeline; in this work we overcome this bottleneck and enhance the end-to-end object detection throughput by 3× with minimal resource cost. Due to the layer-pipelined nature of HPIPE, it can be working on multiple images at the same time and therefore can still exceed the consumption rate of our NMS module. To match the throughput, we can either instantiate N independent NMS modules, each working on a different image, or multi-thread the NMS module to interleave the processing of prediction boxes from N images. We show that multi-threading the NMS unit is much more resource efficient than replicating it, and that a multi-threading factor of 3 (i.e., $N = 3$) is sufficient to match the throughput of the HPIPE backbone CNN.

3 RELATED WORK

While many FPGA-based object detection systems have been introduced in the literature, a number of such systems [6, 20, 32] use software implementations of the NMS pre-processing and NMS stages due to their complex control flow. In contrast, our work introduces a novel pipelined NMS algorithm that allows implementation of the entire end-to-end object detection system on the FPGA. The authors of [34] presented a YOLOv3 [25] object detection system with DarkNet-53 [25] model as its backbone. They implemented an NMS hardware module using bubble sorting for only 3,000 bounding box predictions on an Intel Arria 10 FPGA with a latency of 21 μs just for the NMS pre-processing and NMS components only. In contrast to this implementation, we completely eliminate the need for sorting in our novel NMS algorithm which significantly simplifies the NMS hardware implementation. In addition, our system handles 174,447 bounding box predictions (1,917 boxes for each of the 91 classes) with a total end-to-end latency (including the CNN feature extractor) of 2.1 ms.

An FPGA-based architecture for SSDLite-MobileNet-V2 was also presented in [8]. This model has a **bottleneck residual block (BRB)** layer [27], which stacks a depthwise convolution layer between two pointwise convolution layers to significantly reduce the number of model parameters and compute operations. Their work proposes a fused BRB scheme where the intermediate results of the convolutions are stored in on-chip memory. Their solution implemented on a Xilinx Zynq ZC706 FPGA achieves 65 FPS throughput with 20.3 **mean average precision (mAP)** on the COCO dataset, which does not meet the MLPerf accuracy criteria of achieving at least 22 mAP. Our implementation achieves a significantly higher throughput of 2,167 FPS with an enhanced mAP of 22.8.

The work in [28] presented a custom ASIC implementation of NMS using TSMC 28nm CMOS technology. For 1,000 bounding box predictions, it achieves a latency of 12.7 μs for the NMS pre-processing and NMS stages. This implementation focuses mainly on reducing on-chip memory utilization by storing intermediate box information in a compressed form. In contrast, our work processes bounding boxes as they are generated by the CNN layers in a streamlined fashion without needing intermediate storage for all boxes (i.e., we only store the final selected boxes).

Zhao *et al.* [35] present another custom hardware implementation for NMS on a Xilinx Virtex-7 FPGA. In this work, the authors implement the traditional NMS algorithm (listed in Algorithm 1) in hardware and combine it with an implementation of MnasNet [31] for feature extraction. This implementation achieves 23 FPS throughput with 22.8 mAP accuracy on the COCO dataset. Compared to this work, we present a novel sort-less hardware-friendly NMS algorithm that enables the implementation of a fully-pipelined end-to-end object detection system that achieves ~100× higher throughput at the same mAP.

4 HPIPE FOR BACKBONE CNN ACCELERATION

As shown in Figure 1, SSD-MobileNet-V1 consists of the backbone MobileNet-V1 CNN, SSD convolutions, NMS pre-processing, and NMS. The backbone CNN portion of our object detection system dominates computations; thus, we leverage the high-performance sparsity-aware CNN inference accelerator, HPIPE [11], to accelerate CNN computations. Although we focus on the SSD-MobileNet-V1 model from the MLPerf benchmark suite in this work, changing the backbone CNN to newer MobileNet versions which are supported by HPIPE [11, 29] is also possible without any additional changes. HPIPE outperforms prior FPGA-based CNN inference accelerators and the NVIDIA V100 GPU. For ResNet-50 with a batch size of 1, HPIPE achieves 3.7× higher FPS than the NVIDIA V100 GPU [22], and 10× higher throughput the Microsoft Brainwave [9] and the Intel Deep Learning Accelerator [1] FPGA overlays.

4.1 HPIPE Accelerator Architecture

HPIPE is a dataflow accelerator coupled with a network compiler that constructs a unique hardware unit for every layer in a CNN and stitches them via latency-insensitive FIFO interfaces, and coarse-grained back pressure signals between those layers to ensure correct operation. HPIPE is deeply-pipelined as it allows arbitrary pipelining between the different layers as well as within every layer's hardware unit. It can also exploit weight sparsity by skipping zero-weight computations, which results in a considerable reduction in memory requirements and compute operations.

Customized per-layer hardware leads to high efficiency in utilizing hardware resources, while deep pipelining achieves high operating frequency, leading to high throughput. HPIPE supports a variety of CNN layers, including standard, depthwise, pointwise and sparse convolutions, as well as pooling layers, and multiple activation functions such as **rectified linear unit (ReLU)** and **hyperbolic tangent (tanh)**. The HPIPE compiler takes in a high-level CNN description and automatically generates a set of Verilog modules implementing the accelerator and memory initialization files to store the CNN weights in the FPGA on-chip **block RAMs (BRAMs)**.

Layer Pipelining. In order to allow all layers to perform computations simultaneously, the hardware of a layer in HPIPE prioritizes computing the rows that will enable a downstream layer to start processing. While pipelining CNN computations achieves high performance in HPIPE, it requires the weights of all layers to fit persistently in the on-chip memory of the FPGA. In this work, we accelerate SSD-MobileNet-V1 which contains 4.2M parameters and can easily fit in the on-chip memory of most modern FPGAs. Pipelining the computations acts as an extra dimension of parallelism in the CNN in the sense that all the layers are executing simultaneously on different parts of an image or different images, which significantly contributes to HPIPE's high performance.

Compute Parallelism. HPIPE parallelizes computations across three dimensions in convolutional layers: output width W, input channels C_i, and output channels C_o. While HPIPE fully unrolls the computations across the W dimension, its compiler automatically decides on the C_i and C_o parallelism factors to maximize the pipeline throughput while not exceeding the available FPGA resources.

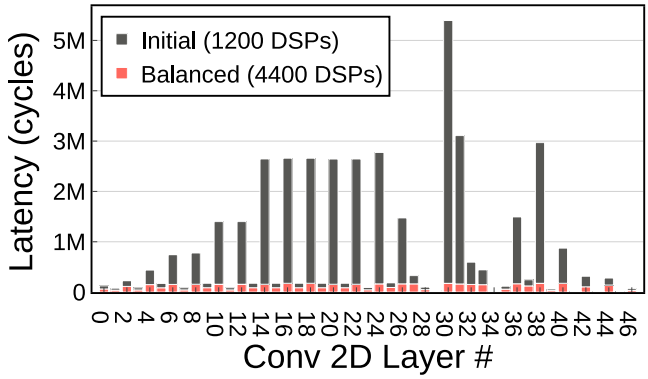


Fig. 2. Throughput balancing between layers in HPIPE.

4.2 HPIPE Compiler

HPIPE generates a unique accelerator for every CNN, which necessitates a high level compiler that translates a CNN into a set of pipelined hardware units. The compiler is written in Python and takes as inputs both a TensorFlow graph describing the network and a budget for each type of FPGA resources. The compiler first parses the graph and performs some optimizations such as merging and swapping nodes to transform the graph to a format consumable by the following stages. Next, the compiler statically splits the hardware resources, mainly DSP blocks, amongst the different layers to maximize throughput. Since HPIPE is a pipeline of layers, the pipeline is as fast as its slowest stage. Thus, the compiler progressively allocates more hardware to the slowest layers in the CNN to balance layer latencies and achieve maximum throughput while making the best use of the available FPGA resources. Figure 2 shows an example of the latencies of the different convolution layers in SSD-MobileNet-V1 before and after the resource allocation stage. Before this stage, the default parallelism is assigned for all the layers and this results in some layers having high latencies (grey bars in Figure 2), which limits the overall throughput of the pipeline. On the other hand, after the compiler allocates more resources to the slower layers, the pipeline is well balanced where all layers have very similar latencies (colored bars in Figure 2), and thus the overall throughput is maximized. Finally, the compiler generates a set of Verilog and memory initialization files that implement a customized accelerator for the given CNN model.

5 NOVEL HARDWARE-FRIENDLY NMS ALGORITHM

Our novel NMS algorithm is shown in Algorithm 2. A traditional NMS algorithm waits until all predicted boxes and class scores are produced for an image, then sorts them based on their confidence values, and finally selects the best boxes and class scores. Instead, our algorithm avoids sorting entirely by considering all incoming box predictions as valid candidates for comparison as soon as they are generated by the SSD convolution layers. This weakens the sequential dependency on the SSD convolution layers; we can pipeline processing of box selection as soon as some box predictions are available, and process them in the order they are generated. Sorting is indeed an expensive operation in hardware, so we improve the compute efficiency of a direct sort with the compare and select logic in Algorithm 2.

Our algorithm uses a list of *selected_boxes* to keep track of the bounding boxes that we select as final output predictions for each image. We replace the sorting routine in the conventional NMS algorithm with iterative comparison and replacement of entries in *selected_boxes*. Each incoming box prediction is associated with a *box_inserted* flag that shows whether it has been inserted in

ALGORITHM 2: Novel NMS Implementation

```
Data: input_boxes for each image, IOU_thr
Result: selected_boxes for each input image
//Pipeline the processing of multiple images
for each image do
    Instantiate selected_boxes list of empty boxes
    for each box in input_boxes do
        box_inserted = False
        for each sbox in selected_boxes do
            if !box_inserted then
                box_inserted = replaceIf(box, sbox, IOU_thr)
            else
                deleteIf(box, sbox, IOU_thr)
    Output selected_boxes list of this image
replaceIf (box, sbox, threshold) is
    if sbox is empty then
        sbox = box
        return True
    IOU = calculateIOU(box, sbox)
    if areSameClass(box, sbox) & IOU > threshold then
        if box.score > sbox.score then
            sbox = box
        return True
    return False
deleteIf (box, sbox, threshold) is
    IOU = calculateIOU(box, sbox)
    if areSameClass(box, sbox) & IOU > threshold & box.score > sbox.score then
        delete sbox
```

the list of selected boxes or not. For each incoming box prediction, we loop over the list of selected boxes and compare the prediction score, class, and IOU of each stored box (if it exists in the list) to that of the new incoming box. If (1) the class matches, (2) the IOU of the new box is higher than the pre-determined threshold, and (3) the prediction score of the new box is better than that of the stored one, then we replace the stored box with the new box and set the *box_inserted* flag to indicate that it is not a candidate for further insertion.

However, if the prediction score of the new box is less than that of the stored one, this means that we already have a higher-confidence box for the same object. Therefore, we do not replace the current entry in *selected_boxes* but still mark the new box as *box_inserted* so it is not considered for insertion anymore. For the remaining loop iterations over the rest of the *selected_boxes* list where *box_inserted* is true, we perform the same comparisons with each of the stored boxes. If the same class, IOU and score conditions mentioned above are satisfied for a stored box, we delete it as we have already inserted a box with a higher prediction score that bounds the same object. Our implementation of the NMS algorithm eliminates the need for sorting box predictions, reducing the algorithmic complexity to $O(CN)$ compared to $O(CN \log N)$ in traditional implementations, where C is the number of classes and N is the number of box predictions per class.

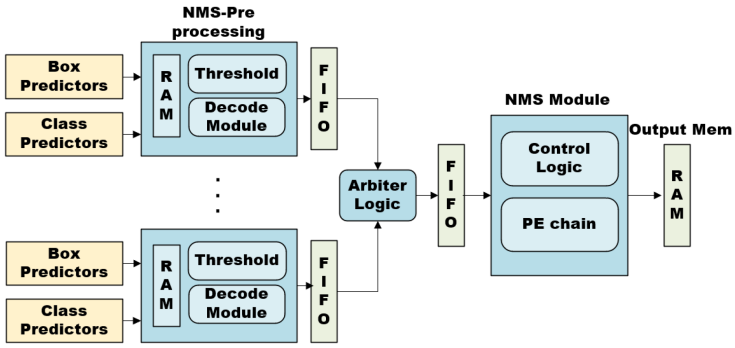


Fig. 3. Block diagram of NMS pre-processing and NMS modules.

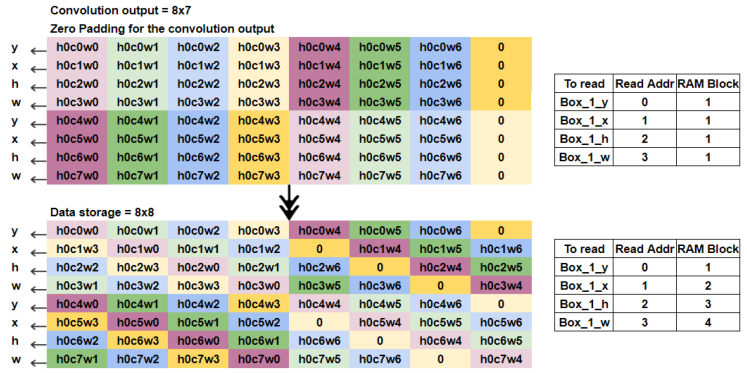
As mentioned earlier, our NMS algorithm processes the generated box predictions one at a time in a streaming fashion as soon as they are produced, instead of waiting for all boxes to be generated first. This approach can be exploited to overlap the execution of NMS with the convolution layers to reduce latency and improve the overall performance. Our earlier work in [2] first introduced this algorithm but it was limited to processing one image at a time, forcing the HPIPE convolution units to remain idle until the NMS processing of a given image finished before starting a new image. In this work, we enhance our NMS algorithm and its hardware implementation to exploit multi-threading by interleaving the processing of multiple images concurrently. This new implementation avoids stalling the convolution units in our end-to-end FPGA system and improves the overall object detection throughput by ~3×.

6 SSD AND NMS HARDWARE IMPLEMENTATION

The overall architecture of the NMS pre-processing and NMS hardware implementation is shown in Figure 3. The box and class prediction convolution layers produce the box coordinate and score predictions corresponding to the 91 different classes of the COCO dataset [2]. Then, these predictions are fed to the NMS pre-processing module which includes a decode module and a threshold module. The decode module contains units for non-linear operations such as sigmoid and exponentiation to convert predictions to actual box co-ordinates and scores. Next, the threshold module discards box predictions with score predictions below a certain pre-determined threshold. Finally, the NMS module consists of top-level control logic and a chain of **processing elements (PEs)** to implement our novel NMS algorithm explained in Section 5.

6.1 Data Formatting

The state-of-the-art CNN accelerator we use in this work generates outputs in the *HCW* format. It produces a complete row of the feature map (W), followed by the same row from all output channels (C) before moving across the height dimension of the feature map (H). This data format allows high parallelism in the HPIPE convolution engine, but does not match the format consumed by our NMS engine which requires all outputs across the C dimension (width, height, x and y offset values) to start processing a box prediction (i.e., *HWC* format). We eliminate the need for any permute operations on box predictions 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 Figure 4, similar to [16]. To achieve this, we concatenate n columns of dummy values (zeroes) to the outputs of the SSD convolution layers, such that $n + W$ is divisible by 4. For each box prediction, we need four pieces of data `{x_offset,y_offset,w,h}`, Therefore, we group the



To read	Read Addr	RAM Block
Box_1_y	0	1
Box_1_x	1	1
Box_1_h	2	1
Box_1_w	3	1

To read	Read Addr	RAM Block
Box_1_y	0	1
Box_1_x	1	2
Box_1_h	2	3
Box_1_w	3	4

Fig. 4. Original (top) vs. modified (bottom) memory data arrangement to store anchor boxes and intermediate convolution results. Cells with same color represents a set of {x,y,h,w} for a bounding box prediction. Each column maps to one RAM block.

data for four box predictions in a 4×4 circular matrix as shown in Figure 4. This allows us to read all four pieces of data for a box prediction in one cycle in parallel from four different RAM blocks. We also use a similar padding plus circulant matrix technique to parallelize access to class scores for each box. 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, simplifying RAM control. In total, we need 104 M20K BRAMs to store both box predictions and class predictions from all SSD convolution layers.

6.2 Thresholding

Typically, thresholding of the score predictions is done at the end of NMS pre-processing (Figure 1). However, in our implementation, we push the thresholding stage before the decoding stage. We directly sample the convolution score predictions based on a modified threshold, and only the boxes with score predictions that exceed the new modified threshold are further processed by the decoding module. This minimizes the storage requirements for intermediate data in the NMS pre-processing. It also reduces the number of non-linear computations executed since only a maximum 200 out of 174, 447 box predictions are expected to be valid candidates for NMS in the COCO dataset. To achieve this, we map the 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.

6.3 Decoding

The {x,y,w,h} values of the box predictions that pass the score threshold are coupled with their corresponding anchor boxes. 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. We convert the constant divisions to multiplications to reduce complexity and use highly-optimized vendor-supplied IPs for arithmetic operations such as multiplications, additions, and exponentiation. We have six different instances of the decoding module that work on the output of the six SSD box prediction convolution layers in parallel. Figure 5 compares the conventional NMS pre-processing architecture to our optimized implementation shown in Figure 5(a) and 5(b), respectively. We avoid the need for multiple read/write operations by eliminating the explicit permute and reshape operations, and we also process less box predictions by pushing the threshold operation before the decode stage.

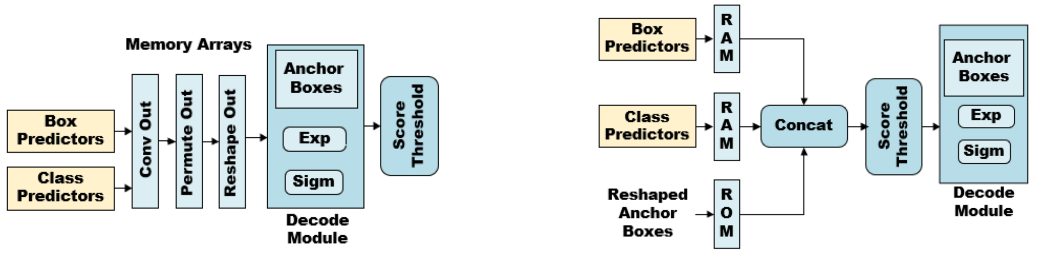


Fig. 5. NMS pre-processing implementations for (a) the conventional scheme and (b) our optimized scheme.

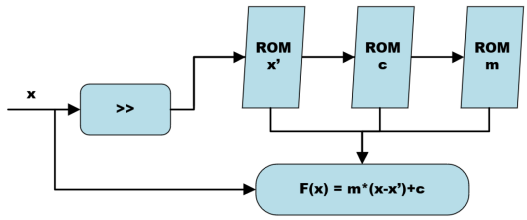


Fig. 6. Hardware implementation of piece-wise linear approximation of non-linear functions.

6.4 Piece-wise Approximation for Non-Linear Operations

The decoding module includes exponential and sigmoid non-linear operations. In our previous work [2], we had implemented these modules using Intel's floating-point IPs. In this work, we use piece-wise linear approximation to implement the non-linear functions. Our hardware implementation for the piece-wise linear approximation is shown in Figure 6. In this approach, non-linear functions are approximated by representing them with a number of line segments. For any given input, we first identify the line segment that the input lies on. Then, we calculate the approximate value of the function for this input using the equation of its corresponding line segment $f(x) = m(x - x') + c$, where x', m and c are the interval start point, slope and y-intercept of the line, respectively. In our implementation, Sigmoid is approximated for the range $[-10, 10]$ with 81 linear segments, and exponentiation is approximated for the range $[-4, 4]$ with 129 linear segments. The full range is split into intervals with a power-of-two width and thus, a simple shift right operation (instead of division) can be used to determine the interval used for approximation. This shift operation produces the address for reading the corresponding x', m, and c values that are pre-computed for each approximated function and stored in a small look-up table. Replacing the floating-point IPs with piece-wise approximation saves 175 DSP blocks, 36 BRAMs and 5,000 ALMs. Additionally, it improves the operating frequency, and reduces the latency of these operations from 72 to only 4 cycles. This reduces end-to-end latency by 0.27 ms with no negative impact on overall object detection accuracy.

6.5 Pipelined NMS

We build a custom NMS module, that implements our novel NMS algorithm explained in Section 5. It consists of a number of chained processing elements (PEs) that accept boxes as inputs and process them to filter out the redundant boxes and store the final boxes using Algorithm 2. For our implementation, we empirically determine that no image in the COCO dataset contains more than

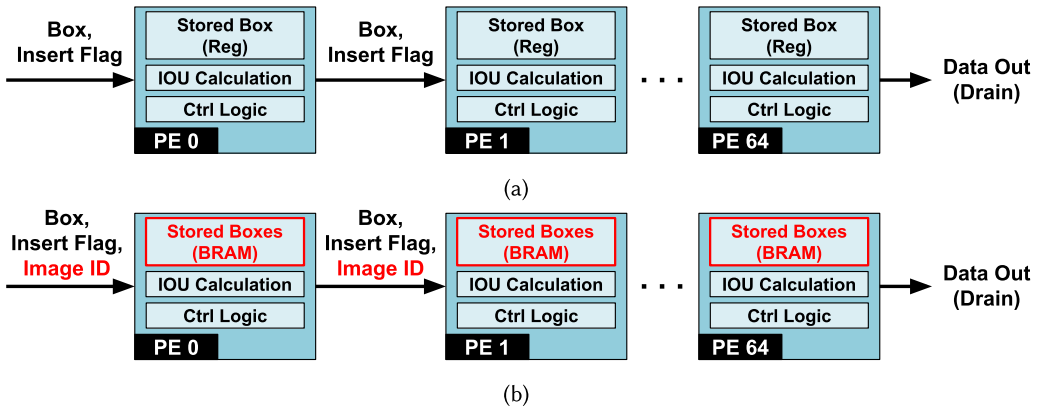


Fig. 7. Custom hardware implementation of our pipelined NMS algorithm (a) from our previous work [2], in comparison to (b) our multi-threaded implementation in this work. All new or changed components are highlighted in red.

65 objects and thus we use a chain of 65 PEs. In our previous work [2], the NMS PEs had registers to store the selected box predictions of a single image as shown in Figure 7(a). Therefore, the backbone CNN running on HPIPE had to stall until NMS processing of a given image is finished to avoid mixing box predictions from different images in the NMS PE chain, degrading the overall system throughput. In this work, we *multi-thread* the NMS execution by adding buffering and control logic to interleave the processing of box predictions from different images, as illustrated in Figure 7(b). In case of the SSD-MobileNet-V1 model, HPIPE can exploit pipeline parallelism across at most three images. Therefore, each PE has a small (BRAM-based) buffer with a depth of 3 to store selected box predictions from three different images. However, our NMS module is parameterizable; it can interleave the processing of more images to match the throughput of the feature extraction CNN implementation if needed. Each PE has its own local control logic to perform all the checks and decide if it should keep the input box, ignore the input box, or delete one of its stored boxes.

The NMS module begins processing as soon as an input box from the SSD convolution and data formatting layers is available, without waiting for all layers to complete for a given image. Each of the input boxes is concatenated with address bits that identify which in-flight image it corresponds to (*Image ID*), and an *Insert Flag* that specifies whether a prediction box was stored/inserted in a previous PE in the pipeline. The control logic at each PE processes the input boxes as follows:

- First, the image address bits are used to fetch the already stored box (if any) from the local buffer.
- Simultaneously, the control logic checks the incoming insert flag and moves to either the insertion or non-insertion state if it is set to 0 or 1, respectively.
- In the insertion state, the input box replaces the stored box if (1) the classes match, (2) the IOU passes the threshold of 0.6, and (3) the new score is higher than the stored one. After a replacement, the *box_inserted* flag is updated to 1. If either the class match or IOU threshold test fails, the stored box is kept and the *box_inserted* flag remains 0. However, if these conditions are satisfied but the score of the stored box is higher, the stored box is kept and the *box_inserted* flag is updated to 1 to reflect that the incoming box should no longer be considered for insertion (i.e., a better box is already stored). Then, the input box is fed to the next PE.

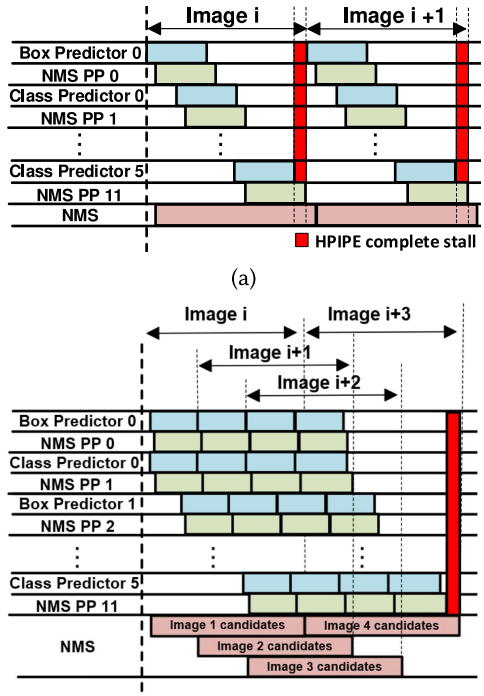


Fig. 8. Object detection timing diagrams for (a) our previous implementation in [2], and (b) the proposed multi-threaded NMS processing in this work.

— In the non-insertion state, the stored box is deleted if (1) the class comparison is true, (2) the score of the input box is greater than the stored box, and (3) the IOU passes the threshold of 0.6. Otherwise, the stored box is kept. Then, the input box is fed to the next PE in the chain.

Since we are not sorting the stored boxes, the final output boxes can be scattered along the PE chain. Therefore, once the NMS module finishes processing all boxes from an image, the output boxes stored in the corresponding location of the PE RAM are drained out of the chain. These will be the final output boxes in the form of $\{x_{min}, y_{min}, x_{max}, y_{max}\}$ that are converted to $\{x_{min}, y_{min}, w, h\}$ to match the output format of the software implementation, and are then written to the system's output FIFO.

7 END-TO-END OBJECT DETECTION SYSTEM

We leverage the same HPIPE CNN accelerator that we implemented in our prior work in [2]. Compared to the original HPIPE implementation from [11], we added support for the SSD convolutions that generate class and box predictions. Moreover, we modified the interfacing modules such that outputs from different convolution layers of HPIPE can be fed to the NMS pre-processing unit.

Figure 8 illustrates the timing diagrams for both our prior work in [2] (Figure 8(a)) and our proposed multi-threaded NMS processing in this paper (Figure 8(b)), respectively. Most of our prior work's operations were overlapping, with 97% of the NMS processing time overlapping at least some execution in the HPIPE convolution engines. However, large gaps still existed between the SSD convolution outputs of different images because the NMS block was designed to handle the computations of only one image. HPIPE on the other hand pipelines multiple images and some

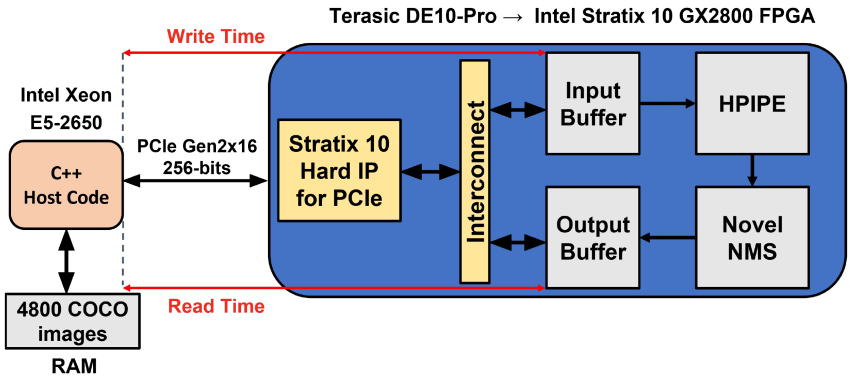


Fig. 9. Overview of the end-to-end object detection system.

of its early SSD layers would stall, as their output buffers are filled with data for image $i+1$ while the NMS unit is still processing image i. These SSD layers would in turn back-pressure prior HPIPE convolution layers, ultimately leading to many idle functional units and some complete pipeline stalls. These factors resulted in a significant under-utilization of the HPIPE hardware, leading to lower performance.

In this work, we alleviated this bottleneck, and as shown in Figure 8(b), the NMS computations of multiple input images are interleaved resulting in better overlap of backbone CNN and NMS processing. We eliminated the gaps between the outputs of the SSD convolutions by multi-threading the NMS module to handle multiple images simultaneously. As shown in the last three rows of Figure 8(b), our NMS unit pipelines three images in order to fully keep up with HPIPE and avoid any stalls. This also necessitates some additional buffering within the NMS processing elements (instead of a single register) to ensure that they can begin processing image $i+1$ without overwriting data for image i that is still in use. The enhancements we make in this work to both the NMS pre-processing and NMS modules come at a low cost of 9,510 ALMs (1%) and 224 BRAMs (2%).

8 RESULTS

8.1 Experimental Setup

Figure 9 shows a high level overview of our end-to-end object detection system. Our end-to-end system runs on a Terasic DE10-Pro board with an Intel Stratix 10 GX2800 FPGA. The board is connected as a PCIe accelerator card to an Intel E5-2650 server with 12 two-way-threaded cores and 94 GB of RAM. We utilize Intel Quartus Prime Pro 19.4 for synthesis, placement and routing, and we use Synopsys VCS to perform RTL-level simulations for functional verification and performance estimation. For power estimation, we use the Quartus Power estimation tool to perform vectorless power estimation. In our setup, the host CPU streams input images to our accelerator and reads back the output predictions via the PCIe link. To verify the functionality of our system, we stream 4,800 images from the COCO validation dataset and measure the accuracy of our system's predictions. We compute our system's performance by measuring the time it takes our system to process all 4,800 images, including the communication time between the host CPU and the FPGA over PCIe.

8.2 Implementation Results

Table 1 lists the resource utilization of the three main components of our system: HPIPE, the NMS pre-processing and the NMS module. Figure 10 shows the chip planner view of our system when

Table 1. Breakdown of FPGA Resource Utilization (NMS-PP: NMS Pre-Processing Module)

	ALMs	DSPs	M20K BRAMs
Full System	618,828 (66%)	5,009 (87%)	7,883 (67%)
HPIPE	504,612 (53.8%)	4,434 (77%)	7,179 (61%)
NMS-PP	35,556 (3.8%)	90 (1.6%)	389 (3.3%)
NMS	60,658 (6.5%)	485 (8.4%)	260 (2.2%)

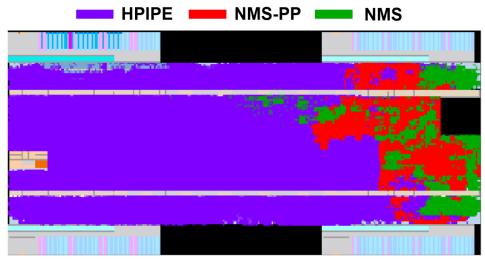


Fig. 10. Chip planner view of our accelerator blocks after being synthesized, placed and routed by the Intel Quartus tool.

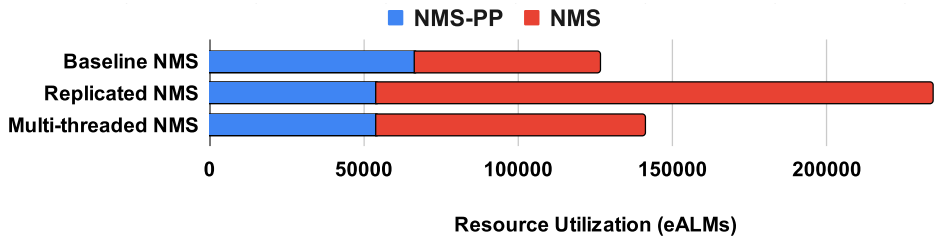


Fig. 11. Resource utilization breakdown of the NMS pre-processing and NMS modules for the baseline from our previous work [2], when replicating the NMS module to match the backbone CNN throughput, and our multi-threaded NMS from this work.

placed and routed by Quartus on the Intel Stratix 10 GX2800 FPGA. Similar to our prior work [2], HPIPE uses most of the FPGA resources compared to the NMS modules, which consume <10% of the available resources. Our system continues to be DSP-bound with nearly 90% DSP utilization.

Figure 11 presents the resource breakdown of the **NMS pre-processing (NMS-PP)** and NMS modules for three design cases: (1) the baseline NMS from our previous work [2], (2) instantiating multiple NMS module to keep up with the backbone CNN throughput, and (3) using the multi-threaded NMS module proposed in this work. The resource breakdown is reported in **equivalent ALMs (eALMs)** assuming BRAMs and DSP blocks are equivalent to 40 and 33 ALMs respectively (based on their relative silicon areas) as in [5]. Since our new NMS pre-processing uses piece-wise approximation units for the non-linear functions, it utilizes 175 fewer DSP blocks compared to our prior implementation in [2], resulting in a 20% reduction in resource utilization for the NMS pre-processing module. Our proposed multi-threaded module can match the throughput of the backbone CNN while being 2× more area efficient compared to the alternative naïve approach of instantiating multiple NMS modules, as shown in Figure 11.

Table 2. Comparison of our Work to Prior FPGA-Based Object Detection Systems on the COCO Dataset

	Fan et al. [8]	Wu et al. [33]	Mobilint [23]	Cai et al. [6]	Anupreetham et al. [2]	This Work
Feature Extraction[1]	M-V2 [27]	M-V1 [13]	M-V1 [13]	M-V1 [13]	M-V1 [13]	M-V1 [13]
Object Detector	SSDLite	SSD	SSD	SSD	SSD	SSD
# Parameters	2.79M	4.2M	4.2M	4.2M	4.2M	4.2M
FPGA Device	Zynq ZC706	Zynq ZU9	Alveo U250	Arria 10	Stratix 10 GX2800	Stratix 10 GX2800
Process Technology	28nm	16nm	16nm	20nm	14nm	14nm
Power (W)	9.9	–	–	–	55	81
Energy Eff. (FPS/W)	6.6	–	–	–	11.1	26.8
Frequency (MHz)	100	333	250	–	350	400
Throughput (FPS)	65	124	410	108	609	2,167
Latency (ms)	15.43	–	10.64	–	2.4	2.13
mAP	20.3	16.2	23.028	16.8	22.5	22.8

[1]M-V1 = MobileNet-V1, M-V2 = MobileNet-V2.

8.3 Performance and Accuracy Results

Despite utilizing most of the available FPGA resources, our system still achieves a high frequency of 400 MHz as we deeply pipeline all our system components. Due to this high operating frequency and the fully-pipelined nature of our system, it achieves 2, 167 FPS and an end-to-end latency of only 2.13 ms. It also scores a mean average precision (mAP) score of 22.8, which is above the 22.0 mAP score threshold set by MLPerf [23] for the COCO validation dataset with SSD-MobileNet-V1. Using the Quartus power analyzer with vectorless analysis, we estimate our system's power consumption to be 81 W, which represents an energy efficiency of 26.8 FPS/W.

8.4 Comparison to Prior Works

Table 2 shows a comparison between our system and other FPGA-based object detection accelerators. Our object detection system achieves the highest throughput, lowest latency, and best energy-efficiency compared to all prior works. This comes at no cost in terms of detection accuracy; in fact, our system scores the second highest mAP among all compared accelerators.

We first compare our new accelerator to our previous accelerator from [2]. Our new accelerator uses the same backbone CNN accelerator along with the multi-threaded version of the NMS unit as well as piece-wise approximation for the non-linear functions. It achieves 3.6× higher throughput than our prior work. The majority of this improvement is attributed to interleaving the processing of multiple images in the multi-threaded NMS unit. In addition, the overall operating frequency of our accelerator increased from 350 MHz in [2] to 400 MHz in this work due to the more efficient implementation of the non-linear functions. The end-to-end latency for a single image is also improved due to the higher accelerator frequency and the reduced latency of the new non-linear function approximations. Although the estimated power consumption increases, our system improves the overall energy-efficiency because of the much higher increase in accelerator throughput.

Another interesting comparison is against Mobilint [23]. They accelerate SSD-MobileNet-V1 on a same generation Xilinx FPGA, but they do not disclose the full details of their implementation. Our new accelerator achieves 5.3× higher throughput and 5× lower latency, while maintaining a similar accuracy (0.2 lower mAP score). Wu *et al.* [33] implemented a specialized high-performance CNN processor on a Xilinx Zynq ZU9 FPGA. Since they are using a smaller device than ours, we scale their reported performance up by a factor of 2.3, which accounts for the difference in the number of utilized DSPs. After scaling, our system still achieves 7.6× higher throughput and 6.6 more mAP points for the COCO validation dataset. Their accuracy degradation can be in part because their accelerator is using an 8-bit fixed-point precision, while ours uses 16-bit fixed-point precision.

Although Fan *et al.* [8] accelerated the SSDLite-MobileNet-V2, which has less computational complexity than SSD-MobileNet-V1, our accelerator achieves a much higher performance and better detection accuracy. The device they are using is an older generation FPGA with fewer resources than ours. However, even if we optimistically scale their performance by a factor of 6.4 to account for the difference in DSP resource counts between devices, we still achieve 5.2× higher throughput and 1.13× lower latency while attaining higher accuracy.

9 CONCLUSION AND DISCUSSION

In this paper, we presented our end-to-end fully pipelined FPGA-based object detection system, which accelerates one of the MLPerf benchmarks, SSD-MobileNet-V1. We introduced a novel NMS algorithm that eliminates the sequential dependency on the preceding convolution layers of the feature extractor and SSD. This allows overlapping the execution of the front-end (feature extractors and SSD layers) and back end (NMS preprocessing and NMS), reducing the overall latency and improving throughput. We implemented a multi-threaded NMS module that exploits our novel algorithm and allows interleaving the processing of multiple images concurrently to match the throughput of our CNN accelerator and prevent any unnecessary stalls. We also implemented piece-wise linear approximation modules for the non-linear operations used in the NMS pre-processing to reduce FPGA resource usage compared to our prior work. Our deeply pipelined FPGA system runs at 400 MHz, and achieves state-of-the-art performance of 2, 167 FPS and an end-to-end latency of only 2.13 ms with a 22.8 mAP score on the COCO validation dataset. These results represent a 5.3× throughput improvement compared to the best performing prior work and the only MLPerf submission on FPGA-based object detection.

We see two vectors for further performance improvements. Firstly, multiple network-attached FPGAs could form a larger hardware accelerator; [14] has recently shown near-linear performance scaling of HPIPE to multiple such FPGAs. Secondly, the recent Stratix 10 NX FPGA includes tensor blocks optimized for deep learning that provide much more multiply accumulate performance [4, 17]. A newer version of HPIPE using these tensor blocks has been recently developed, resulting in more than 8× improvement in throughput on the image classification MobileNet-V1 [29] network. Extending our state-of-the-art object detection system to use multiple tensor-enhanced FPGAs is a very promising direction for future research.

REFERENCES

[1] Mohamed S. Abdelfattah, David Han, Andrew Bitar, Roberto DiCecco, Shane O'Connell, Nitika Shanker, Joseph Chu, Ian Prins, Joshua Fender, Andrew C. Ling, and Gordon R. Chiu. 2018. DLA: Compiler and FPGA overlay for neural network inference acceleration. In *2018 28th International Conference on Field Programmable Logic and Applications (FPL)*. 411–4117. https://doi.org/10.1109/FPL.2018.00077

[2] Anupreetham Anupreetham, Mohamed Ibrahim, Mathew Hall, Andrew Boutros, Ajay Kuzhively, Abinash Mohanty, Eriko Nurvitadhi, Vaughn Betz, Yu Cao, and Jae-sun Seo. 2021. End-to-end FPGA-based object detection using pipelined CNN and non-maximum suppression. In *2021 31st International Conference on Field-Programmable Logic and Applications (FPL)*. 76–82. https://doi.org/10.1109/FPL53798.2021.00021

[3] Andrew Boutros and Vaughn Betz. 2021. FPGA architecture: Principles and progression. *IEEE Circuits and Systems Magazine* 21, 2 (2021), 4–29. https://doi.org/10.1109/MCAS.2021.3071607

[4] Andrew Boutros, Eriko Nurvitadhi, Rui Ma, Sergey Gribok, Zhipeng Zhao, James C. Hoe, Vaughn Betz, and Martin Langhammer. 2020. Beyond peak performance: Comparing the real performance of AI-optimized FPGAs and GPUs. In *2020 International Conference on Field-Programmable Technology (ICFPT)*. 10–19. https://doi.org/10.1109/ICFPT51103.2020.00011

[5] Andrew Boutros, Sadegh Yazdanshenas, and Vaughn Betz. 2018. You cannot improve what you do not measure: FPGA vs. ASIC efficiency gaps for convolutional neural network inference. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)* 11, 3 (2018), 1–23.

[6] Liang Cai, Feng Dong, Ke Chen, Kehua Yu, Wei Qu, and Jianfei Jiang. 2020. An FPGA based heterogeneous accelerator for single shot multibox detector (SSD). In *2020 IEEE 15th International Conference on Solid-State & Integrated Circuit Technology (ICSICT)*. 1–3. https://doi.org/10.1109/ICSICT49897.2020.9278177

[7] François Chollet. 2017. Xception: Deep learning with depthwise separable convolutions. In *Conference on Computer Vision and Pattern Recognition (CVPR)*.

[8] Hongxiang Fan, Shuanglong Liu, Martin Ferianc, Ho-Cheung Ng, Zhiqiang Que, Shen Liu, Xinyu Niu, and Wayne Luk. 2018. A real-time object detection accelerator with compressed SSDLite on FPGA. In *2018 International Conference on Field-Programmable Technology (FPT)*. 14–21. https://doi.org/10.1109/FPT.2018.00014

[9] Jeremy Fowers, Kalin Ovtcharov, Michael Papamichael, Todd Massengill, Ming Liu, Daniel Lo, Shlomi Alkalay, Michael Haselman, Logan Adams, Mahdi Ghandi, Stephen Heil, Prerak Patel, Adam Sapek, Gabriel Weisz, Lisa Woods, Sitaram Lanka, Steven K. Reinhardt, Adrian M. Caulfield, Eric S. Chung, and Doug Burger. 2018. A configurable cloud-scale DNN processor for real-time AI. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. 1–14. https://doi.org/10.1109/ISCA.2018.00012

[10] Ross Girshick, Jeff Donahue, Trevor Darrell, and Jitendra Malik. 2014. Rich feature hierarchies for accurate object detection and semantic segmentation. In *2014 IEEE Conference on Computer Vision and Pattern Recognition*. 580–587. https://doi.org/10.1109/CVPR.2014.81

[11] Mathew Hall and Vaughn Betz. 2020. From TensorFlow graphs to LUTs and wires: Automated sparse and physically aware CNN hardware generation. In *2020 International Conference on Field-Programmable Technology (ICFPT)*. 56–65. https://doi.org/10.1109/ICFPT51103.2020.00017

[12] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 770–778. https://doi.org/10.1109/CVPR.2016.90

[13] Andrew G. Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. 2017. MobileNets: Efficient convolutional neural networks for mobile vision applications. *CoRR* abs/1704.04861 (2017). arXiv:1704.04861http://arxiv.org/abs/1704.04861

[14] Mohamed Ibrahim and Vaughn Betz. 2022. *Extending Data Flow Architectures for Convolutional Neural Networks to Object Detection and Multiple FPGAs*. Master's thesis. The University of Toronto. https://tspace.library.utoronto.ca/handle/1807/123335

[15] Licheng Jiao, Fan Zhang, Fang Liu, Shuyuan Yang, Lingling Li, Zhixi Feng, and Rong Qu. 2019. A survey of deep learning-based object detection. *IEEE Access* 7 (2019), 128837–128868. https://doi.org/10.1109/ACCESS.2019.2939201

[16] Shreyas Kolala Venkataramanaiah, Yufei Ma, Shihui Yin, Eriko Nurvithadhi, Aravind Dasu, Yu Cao, and Jae-Sun Seo. 2019. Automatic compiler based FPGA accelerator for CNN training. In *2019 29th International Conference on Field Programmable Logic and Applications (FPL)*. 166–172. https://doi.org/10.1109/FPL.2019.00034

[17] Martin Langhammer, Eriko Nurvitadhi, Bogdan Pasca, and Sergey Gribok. 2021. Stratix 10 NX architecture and applications *(FPGA'21)*. Association for Computing Machinery, New York, NY, USA, 57–67. https://doi.org/10.1145/3431920.3439293

[18] Tsung-Yi Lin, Michael Maire, Serge J. Belongie, Lubomir D. Bourdev, Ross B. Girshick, James Hays, Pietro Perona, Deva Ramanan, Piotr Dollár, and C. Lawrence Zitnick. 2014. Microsoft COCO: Common objects in context. *CoRR* abs/1405.0312 (2014). arXiv:1405.0312http://arxiv.org/abs/1405.0312

[19] Wei Liu, Dragomir Anguelov, Dumitru Erhan, Christian Szegedy, Scott E. Reed, Cheng-Yang Fu, and Alexander C. Berg. 2015. SSD: Single shot multibox detector. *CoRR* abs/1512.02325 (2015). arXiv:1512.02325http://arxiv.org/abs/1512.02325

[20] Yufei Ma, Tu Zheng, Yu Cao, Sarma Vrudhula, and Jae-sun Seo. 2018. Algorithm-hardware co-design of single shot detector for fast object detection on FPGAs. In *IEEE International Conference on Computer-Aided Design (ICCAD)*.

[21] Jian Meng, Shreyas Kolala Venkataramanaiah, Chuteng Zhou, Patrick Hansen, Paul Whatmough, and Jaesun Seo. 2021. FixyFPGA: Efficient FPGA accelerator for deep neural networks with high element-wise sparsity and without external memory access. In *IEEE International Conference on Field-Programmable Logic and Applications (FPL)*. 9–16. https://doi.org/10.1109/FPL53798.2021.00010

[22] NVIDIA. 2019. NVIDIA Tesla deep learning product performance. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*.

[23] Vijay Janapa Reddi, Christine Cheng, David Kanter, Peter Mattson, Guenther Schmuelling, Carole-Jean Wu, Brian Anderson, Maximilien Breughe, Mark Charlebois, William Chou, Ramesh Chukka, Cody Coleman, Sam Davis, Pan Deng, Greg Diamos, Jared Duke, Dave Fick, J. Scott Gardner, Itay Hubara, Sachin Idgunji, Thomas B. Jablin, Jeff Jiao, Tom St. John, Pankaj Kanwar, David Lee, Jeffery Liao, Anton Lokhmotov, Francisco Massa, Peng Meng, Paulius Micikevicius, Colin Osborne, Gennady Pekhimenk, Arun Tejusve Raghunath Rajan, Dilip Sequeira, Ashish Sirasao, Fei Sun, Hanlin Tang, Michael Thomson, Frank Wei, Ephrem Wu, Lingjie Xu, Koichi Yamada, Bing Yu, George Yuan, Aaron Zhong, Peizhao Zhang, and Yuchen Zhou. 2019. MLPerf inference benchmark. *CoRR* abs/1911.02549 (2019). arXiv:1911.02549http://arxiv.org/abs/1911.02549

[24] Joseph Redmon, Santosh Kumar Divvala, Ross B. Girshick, and Ali Farhadi. 2015. You only look once: Unified, real-time object detection. *CoRR* abs/1506.02640 (2015). arXiv:1506.02640http://arxiv.org/abs/1506.02640

[25] Joseph Redmon and Ali Farhadi. 2018. YOLOv3: An incremental improvement. *CoRR* abs/1804.02767 (2018). arXiv:1804.02767http://arxiv.org/abs/1804.02767

[26] Shaoqing Ren, Kaiming He, Ross B. Girshick, and Jian Sun. 2015. Faster R-CNN: Towards real-time object detection with region proposal networks. *CoRR* abs/1506.01497 (2015). arXiv:1506.01497http://arxiv.org/abs/1506.01497

[27] Mark Sandler, Andrew G. Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. 2018. Inverted residuals and linear bottlenecks: Mobile networks for classification, detection and segmentation. *CoRR* abs/1801.04381 (2018). arXiv:1801.04381http://arxiv.org/abs/1801.04381

[28] Man Shi, Peng Ouyang, Shouyi Yin, Leibo Liu, and Shaojun Wei. 2019. A fast and power-efficient hardware architecture for non-maximum suppression. *IEEE Transactions on Circuits and Systems II: Express Briefs* 66, 11 (2019), 1870–1874. https://doi.org/10.1109/TCSII.2019.2893527

[29] Marius Stan, Mathew Hall, Mohamed Ibrahim, and Vaughn Betz. 2022. HPIPE NX: Boosting CNN inference acceleration performance with AI-optimized FPGAs. In *International Conference on Field-Programmable Technology (FPT)*. IEEE, 1–9.

[30] Amr Suleiman, Yu-Hsin Chen, Joel S. Emer, and Vivienne Sze. 2017. Towards closing the energy gap between HOG and CNN features for embedded vision. *CoRR* abs/1703.05853 (2017). arXiv:1703.05853http://arxiv.org/abs/1703.05853

[31] Mingxing Tan, Bo Chen, Ruoming Pang, Vijay Vasudevan, Mark Sandler, Andrew Howard, and Quoc V. Le. 2019. MnasNet: Platform-aware neural architecture search for mobile. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 2820–2828.

[32] Zixiao Wang, Ke Xu, Shuaixiao Wu, Li Liu, Lingzhi Liu, and Dong Wang. 2020. Sparse-YOLO: Hardware/software co-design of an FPGA accelerator for YOLOv2. *IEEE Access* 8 (2020), 116569–116585. https://doi.org/10.1109/ACCESS.2020.3004198

[33] Di Wu, Yu Zhang, Xijie Jia, Lu Tian, Tianping Li, Lingzhi Sui, Dongliang Xie, and Yi Shan. 2019. A high-performance CNN processor based on FPGA for MobileNets. In *2019 29th International Conference on Field Programmable Logic and Applications (FPL)*. 136–143. https://doi.org/10.1109/FPL.2019.00030

[34] Hui Zhang, Wei Wu, Yufei Ma, and Zhongfeng Wang. 2020. Efficient hardware post processing of anchor-based object detection on FPGA. In *2020 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*. 580–585. https://doi.org/10.1109/ISVLSI49217.2020.00089

[35] Tong Zhao, Lufeng Qiao, Qinghua Chen, Qingsong Zhang, and Na Li. 2020. A hardware accelerator based on neural network for object detection. *Journal of Physics: Conference Series* 1486, 2 (Apr. 2020), 022045. https://doi.org/10.1088/1742-6596/1486/2/022045

[36] Zhengxia Zou, Zhenwei Shi, Yuhong Guo, and Jieping Ye. 2019. Object detection in 20 years: A survey. *CoRR* abs/1905.05055 (2019). arXiv:1905.05055 http://arxiv.org/abs/1905.05055

Received 27 May 2023; revised 27 September 2023; accepted 16 November 2023