Algorithm-Hardware Co-Design of Single Shot Detector for Fast Object Detection on FPGAs

Yufei Ma[1], Tu Zheng[2], Yu Cao[1], Sarma Vrudhula[3], Jae-sun Seo[1]
[1]School of Electrical, Computer and Energy Engineering, Arizona State University, Tempe, USA
[2]College of Computer Science and Technology, Zhejiang University, Hangzhou, China
[3]School of Computing, Informatics and Decision Systems Engineering, Arizona State University, Tempe, USA
{yufeima, yu.cao, vrudhula, jaesun.seo}@asu.edu, zabbit@yeah.net

ABSTRACT

The rapid improvement in computation capability has made convolutional neural networks (CNNs) a great success in recent years on image classification tasks, which has also prospered the development of objection detection algorithms with significantly improved accuracy. However, during the deployment phase, many applications demand low latency processing of one image with strict power consumption requirement, which reduces the efficiency of GPU and other general-purpose platform, bringing opportunities for specific acceleration hardware, e.g. FPGA, by customizing the digital circuit specific for the inference algorithm. Therefore, this work proposes to customize the detection algorithm, e.g. SSD, to benefit its hardware implementation with low data precision at the cost of marginal accuracy degradation. The proposed FPGA-based deep learning inference accelerator is demonstrated on two Intel FPGAs for SSD algorithm achieving up to 2.18 TOPS throughput and up to 3.3× superior energy-efficiency compared to GPU.

KEYWORDS

Hardware Accelerator, FPGA, Neural Network, HW/SW Co-design

1 INTRODUCTION

The recently achieved substantial improvements in speed and accuracy of convolutional neural networks (CNN) for image recognition are now being demonstrated in object detection algorithms. The Single Shot Detector (SSD) [7] algorithm uses VGG-16 [15] as the base feature extractor to predict the bounding boxes and classification probability, and then uses additional convolution layers at the end to predict objects from multi-scale feature maps. With its simplified architecture, the SSD algorithm demonstrates faster performance with higher accuracy, compared to Faster RCNN [13] and YOLO [12]. However, it is still very difficult to directly implement SSD on mobile hardware, e.g. embedded systems and edge devices, to achieve real-time detection with high energy efficiency, because of (1) the large volume of data and operations, (2) the use of complex nonlinear functions, and (3) the highly varying layer sizes and configurations.

To achieve high throughput, high performance GPUs are often used to accelerate the training and inference tasks of CNNs, as they can take advantage of the thousands of parallel cores, operating at high clock frequencies at GHz level, and achieve hundreds of GB/s memory bandwidth. However, their power consumption is too high (>150W) for power and energy constrained platforms. Furthermore, GPUs are best suited for achieving high throughput when processing large batches of images. However, for applications that require very low latency for processing a single image, as in autonomous driving and surveillance, the completion of detection must be done at the speed of incoming data stream, which degrades GPUs' performance and energy-efficiency substantially.

On the other hand, field-programmable gate arrays (FPGAs) have gained increasing interests and popularity to accelerate the inference tasks of CNNs, due to their (1) high degree of reconfigurability, (2) faster development time compared to application-specific integrated circuits (ASICs) [10][14], (3) good performance, and (4) superior energy efficiency compared to GPUs [1][17][8]. The high performance and efficiency of an FPGA can be realized by synthesizing a circuit that is customized for a specific computation to directly process billions of operations with the customized memory systems. For instance, hundreds to thousands of digital signal processing (DSP) blocks on modern FPGAs support the core CNN operations, e.g. multiplication and addition, with high parallelism. Dedicated data buffers between external DRAM memory and on-chip processing elements (PE) can be designed to realize the preferred dataflow by configuring tens of MByte on-chip block random access memories (BRAM) on the FPGA chip.

Directly implementing the original SSD algorithm onto an FPGA may cause low utilization of the available computation resources and consequently result in low performance and efficiency. To address this issue, we tailor the SSD300 algorithm for efficient hardware realization (henceforth referred to SSD300_HW), and employ low precision fixed-point data with dynamic quantization for inference. The process of customizing SSD300 is illustrated in Figure 1, where the complex non-linear functions are removed and varying convolution configurations are unified. These reduce the FPGA resource cost and improve the performance. The proposed FPGA-based deep learning inference accelerator is designed for two Intel FPGAs – Arria 10 and Stratix 10, achieving 1.03 TOPS and 2.18 TOPS throughput for SSD300, respectively, and up to 6.3× less power consumption and 3.3× higher energy-efficiency compared to a high-end GPU.

2 RELATED WORK

Recent FPGA works on hardware acceleration of CNN inference have demonstrated throughput improvement from 62 GOPS [20] to 1,382 GOPS [1], while also significantly improving the energy efficiency when compared to GPU based implementations [19][3]. Efforts have been made to reduce the gap between the rapid development of deep learning algorithms and the long design time for FPGA

ICCAD'18, 2018, San Diego, CA, USA
2018. ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

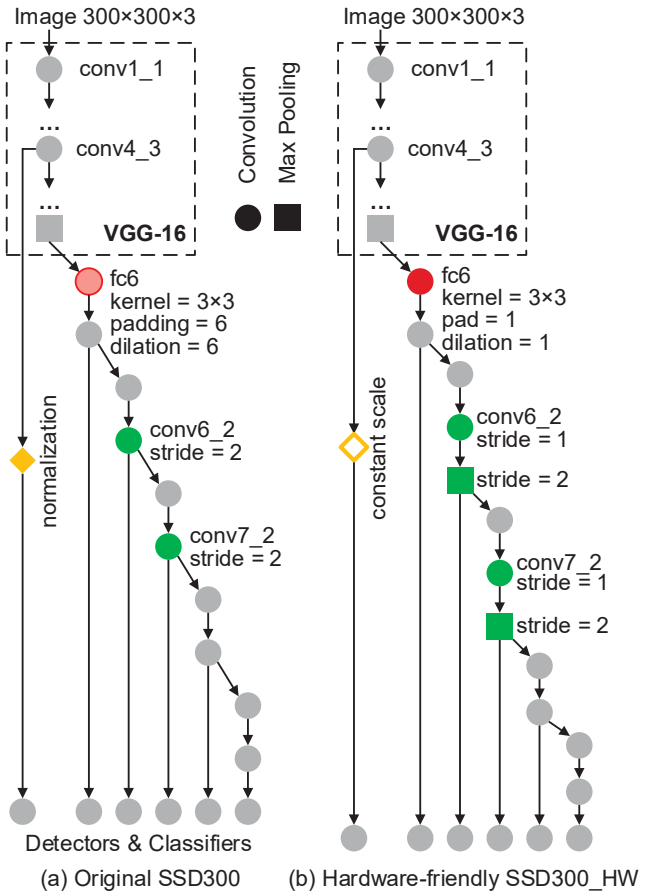

Figure 1: Customization of SSD300 to be hardware-friendly SSD300_HW by (1) replacing dilated convolution, (2) using constant scale instead of normalization and (3) using uniform convolution stride, where the algorithm customizations are highlighted by different colors.

hardware implementation. Several FPGA-based frameworks or compilers have been proposed to automatically map several different CNN algorithms, e.g. AlexNet [6], VGG [15] and ResNet [4], onto FPGA hardware [3, 8, 19]. To speed up the FPGA implementation, approaches based on high-level synthesis (HLS) techniques and the use of OpenCL, are becoming increasingly popular, due to their easy programmablity and reduced design time [1, 16, 17, 20]. However, the conventional design methodology that relies on manual register-transfer level (RTL) allows much finer level of optimization of the hardware, resulting in higher performance and energy efficiency [9, 11]. This is the approach adopted in the present work. In contrast to other applications of CNNs, there have been relatively fewer works focused on FPGA-based object detection. The most recent examples inlcude the Xilinx Zynq FPGA [21], which has been used to implement YOLO (GoogLeNet) and the Faster RCNN (VGG16). The reported latency of these two designs are 744 ms and 875 ms per image respectively, and cannot be considered to be in real-time.

3 HARDWARE ACCELERATION OF CNN ON FPGA

More than 90% of the operations in a CNN are covolution, which is essentially accumulation of large numbers of multiplications (MAC) along different dimensions of the feature and kernel maps [4, 7, 15]. This provides a large design space for exploring parallelism. Modern FPGAs provide hundreds to thousands of DSP blocks to support hardware multiplication operations and megabytes of on-chip memories to buffer data between the on-chip PEs and the external memory, e.g. DRAM. The increasing scale and complexity of CNN algorithms make the goal of maximizing the utilization of the FGPA resources to achieve high throughput and energy efficiency a very challenging task.

3.1 Loop Unrolling for Parallel Computations

The large number of convolution operations require full utilization of the limited FPGA computation resources, e.g. DSP and reconfigurable logic, to maximize the parallel MAC operations. Loop unrolling [9, 20] is used to direct the parallelism along different convolution loops. To reduce the complexity of hardware design, in this work, unrolling of identical loops is applied for all the convolution layers to achieve a uniform mapping across PEs. As the sizes of feature and kernel maps vary dramatically across different convolution layers, we must unroll loops with large dimensions to support high parallelism. Therefore, we choose to parallelize convolution of multiple features inside one input feature map with multiple output kernel maps, and sequentially slide within a kernel window and across all the input feature maps. By this means, the MAC unit with one multiplier followed by an accumulator, as shown in Figure 4, is employed as a PE to perform the convolution operation. The partial sums are accumulated within one MAC unit, and both the input features and kernel weights are reused by multiple MAC units. To implement MAC units in our design, we exploit the built-in multipliers of the DSP blocks in the FPGA.

3.2 Loop Tiling for Memory Storage and Transaction

Since the on-chip memory of an FPGA is typically insufficient to store all the weights and intermediate features for large scale CNN algorithms, loop tiling [9, 20] is applied to divide a large CNN layer into multiple small tiles, which can be accommodated by the on-chip buffers. However, if the tile or buffer size is too small, weights or features may need to be read more frequently from the DRAM. This can dramatically increase the number of DRAM accesses and lead to higher transaction delay and energy consumption. Limited by the external memory bandwidth, the dual buffer structure, as shown in Figure 2, is employed in this work to overlap the external memory transaction delay with the computation latency to improve the overall throughput.

3.3 Accelerator Architecture

The overall FPGA-based CNN inference acceleration system is shown in Figure 2. Both the weights and features are stored in external DRAM due to their large storage requirement. After the acceleration starts, weights and features are loaded into the weight

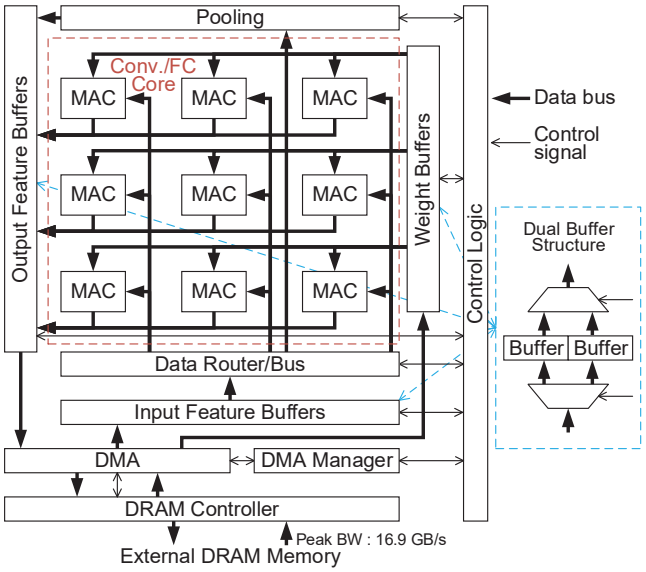

Figure 2: The overall FPGA-based CNN acceleration system, including PE arrays of MAC units, on-chip buffers and external memory interface.

and input feature buffers from DRAM, respectively. The data router is designed to reshape the data fetched from buffers and directs them into the MAC units in a proper dataflow, and it also adapts the dataflow to different convolution sliding strides and zero padding. The output results of MAC units are sent to the output feature buffer and finally written back to external memory to be used as the input for the following layer. The control logic generates read/write addresses and governs the sequential tile-by-tile and layer-by-layer execution with varying layer sizes so that the MAC computation units are serially reused by all the convolution layers.

4 ALGORITHM CUSTOMIZATION FOR FPGA

Unlike software (CPU-GPU) implementations, direct hardware implementation normally favors performing massive numbers of linear computations in parallel, and with a uniform dataflow, as this maximizes the utilization of the hardware resources and reduces the complexity of the control logic. Therefore, it is necessary to tailor the original software implementation to benefit their hardware implementation, while maintaining sufficient accuracy. The modification methods and their corresponding accuracies are shown and summarized in Figure 1 and Table 1.

4.1 Dilated Convolution

To speed up the training and inference time in the original SSD algorithm [7], the fully connected layers, e.g. fc6 and fc7, are converted to convolution layers. In addition, the fc6 layer is implemented as dilated convolution to expand the receptive field without loss of resolution or coverage [18]. However, the change of the computation pattern in convolution makes the dataflow into the PEs significantly different from the original convolution, which requires new data bus and control logic in hardware.

One solution is to implement the dilated convolution as original convolution, filling the intervals inside the kernel window with zeros. The cost of this increases redundant computation. In SSD, the configuration of fc6 is kernel size = 3, dilation = 6, and zero-pad = 6. These can be implemented as a normal convolution with kernel size = $3 + 2 \times (6 - 1) = 13$, dilation = 1 and zero pad = 6. By this means, the number of fc6 operations is dramatically increased from 3.4 GOP to 64 GOP. This is even larger than the total number of operations in the original SSD algorithm, i.e. 62 GOP, and is obviously unacceptable.

Another solution is to change the dilated convolution into a normal convolution directly and make the convolution configurations uniform with other layers. Therefore, we set fc6 to be kernel size = 3, dilation = 1, and zero-pad = 1. This makes the output feature map size have the same number of operations. After retraining the SSD model, the mAP of SSD300 with the modified fc6, e.g. SSD300_1, is 77.34% as shown in Table 1. This is even slightly better than the original one as 77.30%. By this means, we can keep using the existing data bus and control logic to implement fc6 without any performance penalty.

4.2 Normalization

Since conv4_3 in SSD has a different feature scale compared to the other layers, [7] applies the L2 normalization combined with scale at each location in the feature map and learn the scale during back propagation. The normalization operation of conv4_3_norm in SSD is expressed as:

$$out(x, y, m) = \frac{scale(m) \times input(x, y, m)}{\sqrt{\sum_{m=1}^{M} input(x, y, m)^2}}, \quad x \in [1, X], y \in [1, Y], m \in [1, M], \tag{1}$$

where X and Y are the feature map width and height, respectively, and M is the number of feature map channels. Computing Equation 1 requires sum of squares, square root and division operations, which are complex in hardware and require large number of logic resources. Instead of directly implementing hardware for these computations, we can alternatively approximate this nonlinear function by using lookup tables to store limited points of the function, which also requires significant amount of on-chip memory and logic. Since conv4_3_norm is only used to scale the feature values to be the same level as other layers, we directly scale all the conv4_3 features with a constant number during training and use the same scale value for inference. As shown in Table 1, we have tried several scale values, e.g. 0.01 for SSD300_2, 0.015 for SSD300_3, and 0.02 for SSD_4, and find that 0.015 scale results in the best mAP of 77.88% (SSD300_3), which is even better than the original 77.30%. By this means, we can directly scale all the features of conv4_3 by a constant number, which significantly simplifies the control logic and reduces the required hardware computing resources.

4.3 Convolution with Different Sliding Strides

Different sliding strides and zero padding in convolutions lead to different dataflow of input features into the PEs. This requires different databus and control logic to govern the dataflow and ensure

Table 1: Experiments of SSD customization for hardware inference with mAP tested on VOC07+12 test database [2]

Model	Dilated Conv. (fc6)	Norm	Constant Scale	Different Conv. Strides	mAP
SSD300	√	√	-	√	77.30%
SSD300_1	×	√	-	√	77.34%
SSD300_2	√	×	0.01	√	77.81%
SSD300_3	√	×	0.015	√	77.88%
SSD300_4	√	×	0.02	√	77.19%
SSD300_5	√	√	-	×	77.41%
SSD300_HW	×	×	0.015	×	77.10%

that the proper input data are continuously fed into PEs without idle clock cycles. Therefore, the hardware design favors regular and uniform convolution structures, e.g. VGG-16, to reduce the design efforts and complexity as well as the required hardware resources. In the original SSD, conv6_2 and conv7_2 use stride of 2 to scale down the output feature map size for multi-scale detection and all other convolution layers have stride of 1, which is not favored by hardware design. Therefore, we change the stride of conv6_2 and conv7_2 to be 1 and add a subsequent max pooling layer with stride of 2 to downsample the feature map. The additional max pooling layers reuse the existing hardware module for the previous pooling layers, which does not add overhead to the hardware resources. This modification adds about 0.64 GOP operations (≈ 1.0% of the total SSD operations) and does not affect the overall performance noticeably. The accuracy of this modification is shown in Table 1 to be 77.41% as SSD300_5.

4.4 Hardware-friendly SSD300_HW

After collectively applying all the aforementioned modifications of (1) removing dilated convolution, (2) using constant scale instead of normalization and (3) employing uniform convolution stride, we obtain the final hardware-friendly SSD300_HW as shown in Table 1 with mAP of 77.10%, which is slighlty lower than the original SSD300 by 0.20%.

5 FPGA INFERENCE WITH LIMITED PRECISION

Although 32-bit floating point precision may be required for the training phase, such a high precision is not necessary for inference, and thus most of the hardware inference works to date use fixed-point data precision without significant loss of accuracy [9–11, 14, 16, 17]. Using data with low precision reduces considerably the requirement of on-chip memory capacity and external memory bandwidth. It also improves the hardware efficiency and performance by allowing the use of fixed-point arithmetic operations, which demands significantly fewer FPGA computing resources, e.g. logic and DSP, compared to floating-point operations.

5.1 Fixed-point Data Representation

Quantization is one of the most commonly used method to convert floating-point represented real numbers into fixed-point format with lower precision. The bit width of a signed fixed-point number (bit_total) is comprised of one sign bit (bit_sign), integer bits (bit_int) and fractional bits (bit_fra) as shown by Equation 2:

$$bit\_total = bit\_sign + bit\_int + bit\_fra. \tag{2}$$

In conventional fixed-point hardware implementation, the decimal point is fixed, and defines the portion between the integer and fractional bits of all the numbers. The integer bit of all the numbers ($\boldsymbol{x}$) is determined as:

$$bit\_int = \lceil \log_2 \max(|\boldsymbol{x}|) \rceil. \tag{3}$$

If bit_int is larger than $bit\_total - 1$, it causes overflow error due to the large scale of the numbers. If bit_int is smaller than $1 - bit\_total$, there is underflow problem due to the small scale of the numbers, which may lead to significant precision loss. The fixed-point integer number $\boldsymbol{X}$ can be obtained by rounding to the nearest integer as Equation 4:

$$\text{Rounding}: \boldsymbol{X_R} = \left\lfloor \boldsymbol{x} \times 2^{bit\_fra} \right\rceil, \tag{4}$$

or truncated to the largest previous integer as Equation 5, which is easier to implement in hardware by right shifting or discarding the least significant bits (LSB):

$$\text{Truncation}: \boldsymbol{X_T} = \left\lfloor \boldsymbol{x} \times 2^{bit\_fra} \right\rfloor. \tag{5}$$

5.2 Dynamic Quantization

Due to the large range and variance in the data in a given CNN algorithm, the conventional fixed-point representation has to increase bit_total to solve the issue of overflow and underflow resulting in higher usage of hardware resources, e.g. memory and logic.

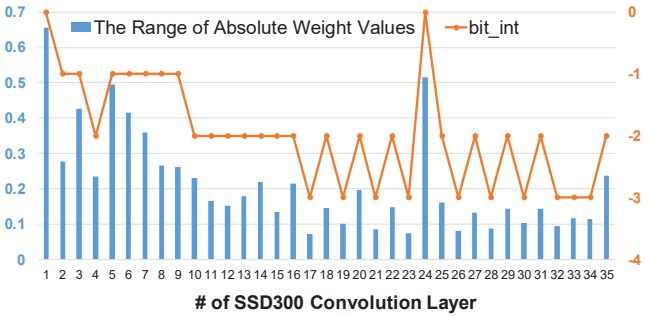

Figure 3: The range of absolute values of each convolution layer's kernel weights in SSD300 and their corresponding bit_int.

Example	bit_total	bit_sign	bit_int	bit_frac	Real number	Fixed-point integer
Input Pixel	16	1	13	2	6789.625	27158
Weight	8	1	-2	9	0.203125	104
Output Pixel	16	1	12	3	1379.142	11033

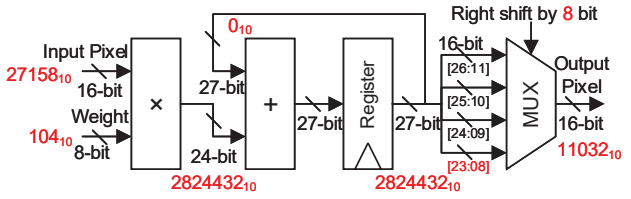

Figure 4: The design of one MAC unit with dynamic quantization for convolution and FC operations, where the multiplier is implemented by DSP and the adder is implemented by logic.

To overcome this problem, we employ the *dynamic quantization* method [1, 9, 11, 14] to use fixed-point representation within one layer and vary the decimal point across different layers. This exploits the characteristic that the range of data in one layer is much smaller than the range across all the layers as shown by Figure 3. By this means, all the weights or all the features of one layer share the same exponent, e.g. bit_fra, and have at most $bit\_total - 1$ bits of significand, whereas in a floating-point representation each number has its own exponent and fixed bits of significand. The constraint on the bit_int is relaxed to be any integer number, which allows for a wide range of values. For example, if the maximum absolute value of the weights in one layer is 5678_{10} and $bit\_total = 8$, then $bit\_int = 13$ according to Equation 3 and $bit\_fra = 8 - 1 - 13 = -6$. For one weight in this layer, e.g. $x = 2345.625_{10} = 100100101001.101_2$, its corresponding fixed point number after truncation is $X_T = 36_{10} = 100100_2$ by Equation 5, where we have 6 bit significand with the rest LSB discarded.

5.3 Dynamic Quantization on Hardware

In Intel Arria 10 and Stratix 10 FPGAs, there are limited number of DSP blocks to implement multipliers for convolution operations. One DSP block can support either one single-precision floating-point multiplier or two 18-bit × 18-bit fixed-point multipliers. Based on this, fixed-point arithmetic can potentially achieve at least twice the throughput compared to floating-point arithmetic by more efficiently utilizing the available DSP resources [1]. Moreover, lower precision also benefits the memory transactions to reduce the memory access delay and energy cost.

The design of the MAC units to compute convolution and fully-connected layers are shown in Figure 4 with an example to illustrate dynamic quantization. The inputs, weights and outputs are assumed to have bit_total = 16, 8, and 16 and bit_fra = 2, 9, and 3, respectively, as listed in the table inside Figure 4, where the multiplier has 24 (=16+8) bit of outputs and the adder has 27 bit of outputs with 3 redundant bit for accumulation. Since the data range of weights and features in one layer could be quite different, we set independent exponents or bit_fra for weights and features. The different bit_fra of inputs and outputs is caused by the different feature value ranges between different layers, or the decimal point is floated across different layers. In order to fit the 27 bit MAC output into the same number of bits as the 16 bit input, the 27 bit output must be truncated or right shifted. The number of bit to be right shifted (bit_right) is determined by the bit_fra of input, weight and output:

$$bit\_right = bit\_fra_{input} + bit\_fra_{weight} - bit\_fra_{output}. \quad (6)$$

In the example inside Figure 4, the MAC output 2,824,432 needs to be right shifted by 8 (= 2 + 9 − 3) bits or discarding the 8 LSB to be 11032, which is different from 11033 in the table because of the error caused by truncation and limited precision. Since different layers may have different bit_right, a multiplexer is needed at the end to choose different truncated outputs with different bit_right, which is the only hardware overhead caused by dynamic quantization compared to the static fixed point design. For the inference phase, the weights are pre-trained so that we can calculate bit_fra and bit_int of each layer off-line before execution as shown in Figure 3. Then, all the weights are dynamically quantized by rounding to be fixed point integer numbers as in Equation 4 and stored in external DRAM to be used by the hardware CNN accelerator. The ranges of feature values are obtained from testing the overall dataset, and then bit_fra and bit_int of each layer are calculated. By this means, the bit_right of each layer is calculated by Equation 6 to control the multiplexer inside the MAC unit.

The detection accuracies of floating-point arithmetic, dynamic quantization and conventional fixed point arithmetic on VOC07+12 test dataset are compared in Table 2 for original SSD300 and hardware friendly SSD300_HW. 16-bit precision with dynamic quantization can provide the same level of accuracy compared with single-precision floating-point arithmetic for both original and modified SSD algorithms. For conventional fixed-point arithmetic, bit_int has to be large enough to cover the wide range of data of the entire SSD algorithm leading to fewer bit_fra and lower precision. Compared with weights, features are more sensitive to precision and require more bit width. Since 8-bit weights do not reduce the accuracy significantly and can save a considerable amount of logic and memory usage, we decide to use 8-bit weights and 16-bit features with dynamic quantization.

6 EXPERIMENTS

6.1 Experimental Setup

CPU and GPU: The baseline CPU used in the experiment is Intel Core i7-5930K with 6 cores, and the GPU is NVIDIA GeForce GTX 1080 Ti. Their detailed specifications are listed in Table 4. The software deep learning framework we used is Caffe [5].

FPGA: The two Intel FPGAs used in the experiment are Arria 10 GX 1150 and Stratix 10 GX 2800. The main FPGA computation resources are DSP blocks and adaptive logic modules (ALM). The main memory resource on FPGA chip is the block random-access memory (BRAM) in terms of M20K with each M20K having 20 Kbit capacity. There are 1,518/5,760 DSP blocks, 427K/933K ALMs, and 2,713/ 11,721 M20K BRAMs on the used Arria 10 and Stratix

Table 2: The accuracies of original SSD300 and hardware-friendly SSD300_HW with different inference precisions are compared on VOC07+12 test set, and the highlighted precision is chosen for FPGA implementation.

Model	Weight Precision	Pixel Precision	Dynamic Quantization	mAP
SSD300	FP-32	FP-32	-	77.30%
SSD300	16	16	√	77.29%
SSD300	8	16	√	77.06%
SSD300	16	8	√	59.36%
SSD300	8	8	√	58.82%
SSD300	16	16	×	75.21%
SSD300	8	16	×	74.68%
SSD300_HW	FP-32	FP-32	-	77.10%
SSD300_HW	16	16	√	77.11%
SSD300_HW	**8**	**16**	√	**76.94%**
SSD300_HW	6	16	√	35.12%
SSD300_HW	16	8	√	53.60%
SSD300_HW	8	8	√	53.23%
SSD300_HW	16	16	×	74.85%
SSD300_HW	8	16	×	74.10%

10, respectively. The underlying FPGA boards for Arria 10 and Stratix 10 are Nallatech 385A and Stratix 10 FPGA Development Kit, respectively, and both are equipped with DDR3 DRAM with peak memory bandwidth of 16.9 GB/s.

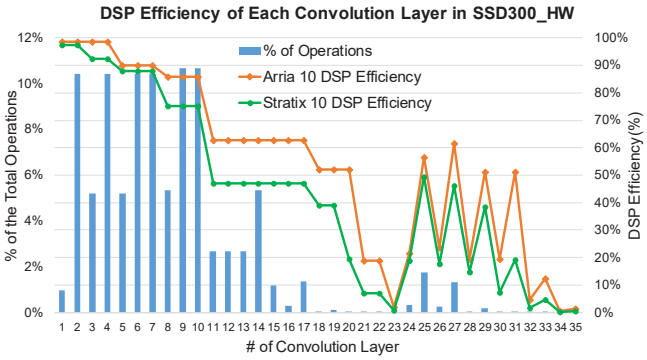

Figure 5: The DSP efficiency of each convolution layer in SSD300_HW is used to measure the match degree between parallel computation scheme and the feature maps.

6.2 Discussion of Results

6.2.1 Parallel Computation Efficiency. To achieve better performance with higher parallelism, we attempt to maximize the usage of DSP blocks for the MAC operations. Each DSP supports two fixed-point multipliers in two MAC units. Constrained by the number of available DSP blocks, we set the number of MAC units on Arria 10 and Stratix 10 to be 3,072 (= 8 × 6 × 64) and 8,192 (= 16 × 8 × 64), respectively. This means 8×6 or 16×8 features within the same output feature map are processed in parallel and such 64 output feature

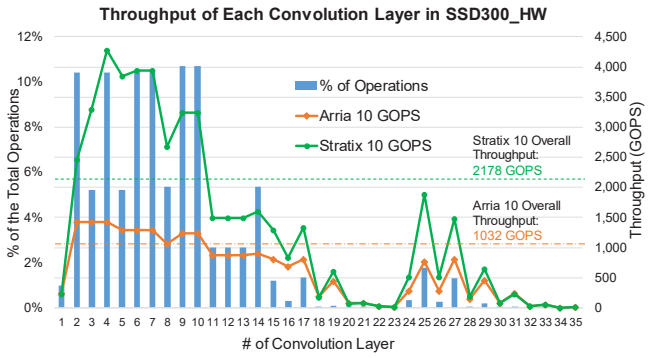

Figure 6: The throughput of each convolution layer in SSD300_HW is constrained by the DSP efficiency and memory bandwidth.

maps are simultaneously computed. Since the feature map sizes and output channel numbers vary significantly across different layers in SSD, the parallel degree and shape may not perfectly match the feature map size and dimension, which causes inefficient utilization of DSP blocks or MAC units. Therefore, the DSP efficiency [17] is defined to measure how well the parallel computation scheme matches the feature maps:

$$DSP\_eff. = \frac{\#\ \ effective\ \ ops.}{\#\ \ actual\ \ performed\ \ ops.}. \tag{7}$$

The DSP efficiency of each convolution layer is shown in Figure 5. The first several layers in SSD300 have large feature map sizes, e.g. 300 × 300 and 150 × 150, so that the parallel dimension can easily fit into the feature maps. The layers at the end for multi-scale detection have much smaller feature maps, e.g. 10 × 10 and 5 × 5, which leads to considerable degradation of DSP efficiency. Fortunately, the first several layers account for most of the total operations as shown in Figure 5, which makes the overall DSP efficiency still high as 81.8% on Arria 10 and 71.5% on Stratix 10. Stratix 10 has larger parallel degrees than Arria 10, which makes it more difficult to match all the feature maps and results in lower DSP efficiency.

Table 3: Comparison of SSD300_HW with baseline SSD300_3 on Arria 10 and Stratix 10 FPGAs

FPGA	Arria 10 GX 1150		Stratix 10 GX 2800	
Model	SSD 300_3	SSD 300_HW	SSD 300_3	SSD 300_HW
Precision	8-16 bit	8-16 bit	8-16 bit	8-16 bit
mAP	77.45%	76.94%	77.45%	76.94%
Clock (MHz)	200	240	240	300
# MAC units	3,072	3,072	8,192	8,192
DSP Block	1,518	1,518	4,370	4,363
Logic (ALM)	220K	175K	618K	532K
BRAM (M20K)	2,586	2,581	3,862	3,844
Latency (ms)	72.2	61.4	35.2	29.1
GOPS	876	1,032	1,798	2,178

Table 4: SSD300 Inference Performance and Efficiency Comparison on Different Platforms with Batch Size = 1

Platform	Intel Core i7-5930K CPU	NVIDIA GeForce GTX 1080 Ti GPU	Intel Arria 10 GX 1150 FPGA	Intel Stratix 10 GX 2800 FPGA
Technology	22 nm	16 nm	20 nm	14 nm
Clock Frequency	3.50 GHz	1.48 GHz	240 MHz	300 MHz
Max. Memory BW	68 GB/s	484 GB/s	16.9 GB/s	16.9 GB/s
Precision	FP-32 bit	FP-32 bit	fixed 8-16 bit	fixed 8-16 bit
mAP of SSD300	77.30%	77.30%	76.94%	76.94%
Latency/Image (ms)	3,272.2	32.58	61.45	29.11
Overall Throughput	19.5 GFLOPS	1,956 GFLOPS	1,032 GOPS	2,178 GOPS
Power (W)	140	250	40	100
Energy/Image (J)	458	8.1	2.4	2.9
Efficiency (GOP/J)	0.14	7.82	25.8	21.8

Note that we employed the SSD300 algorithm with data augmentation, which shows 77.3% mAP but GPU performance was not reported in [7]. For SSD300 without data augmentation, 46 fps was reported for Titan X GPU, but mAP was degraded to 74.3%.

6.2.2 Throughput. The throughput of each convolution layer in SSD300_HW, which is determined by the number of MAC units, DSP efficiency, buffer sizes, and external memory bandwidth, is shown in Figure 6. If there is unlimited memory bandwidth, the shape of the throughput curve in Figure 6 should match the DSP efficiency curve in Figure 5. With limited memory bandwidth, the memory access delay may be larger than the computation delay in some layers, or these layers are memory-bounded. For example, the first convolution layer (conv1_1) is memory bounded for both Arria 10 and Stratix 10. Although Stratix 10 can compute the MAC operations faster, it can only achieve the same throughput as Arria 10, because both of them are memory bounded with the same memory bandwidth. With higher computation speed and the same memory bandwidth, Stratix 10 encounters memory-bounded situations more often than Arria 10, which poses limitations on the throughput improvements of Stratix 10. With 8,192 MAC units operated at 300 MHz, the theoretical maximum throughput of Stratix 10 is 4,915 GOPS, which is 3.3× larger than the Arria 10 maximum throughput of 1,474 GOPS. However, Stratix 10 achieves 2.1× enhancement of throughput over Arria 10 due to the limited memory bandwidth and lower DSP efficiency.

6.2.3 SSD300_HW vs. Baseline SSD300_3. To evaluate the effect of tailoring SSD300 to achieve an efficient hardware implementation , e.g. SSD300_HW, we also implement SSD300_3 as in Table 1, where dilated convolution (fc6) and different convolution strides are unchanged. The detailed comparison results are listed in Table 3, including resource utilization and throughput. Due to the special dataflow of dilated convolution, dedicated control logic and data path router are designed in SSD300_3, which need extra design time and efforts. To support convolution layers with strides of two, additional data buses are used to feed proper data into the PEs. Therefore, SSD300_3 implementations on Arria 10 and Stratix 10 consume about 26% and 16% more logic elements (ALMs) than SSD300_HW, respectively, as in Table 3. Even worse, the additional data buses tighten the critical path and decrease the operating frequency leading to 1.17× and 1.21× throughput reduction compared to SSD300_HW, on Arria 10 and Stratix 10, respectively. The complex nonlinear function involved in the normalization of conv4_3 is expected to require considerably more design efforts and hardware resources that may result in even lower performance. Hence we did not continue to implement normalization for the baseline design. The example detection results of SSD300_HW are shown in Figure 7.

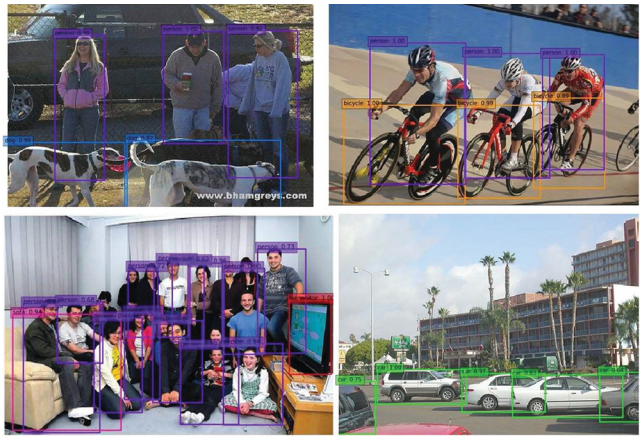

Figure 7: Example detection results of SSD300_HW.

6.2.4 FPGA vs. CPU, GPU. In Table 4, we compare our FPGA-based inference engine with CPU and GPU platforms, for SSD300 implementation. Many latency-critical inference applications, e.g. autonomous drive and surveillance, require the completion of detection at the speed of incoming data stream. Although the high batch size can improve the throughput by sharing the memory transfer delay, it worsens the latency between one input image and its detection result. Therefore, we set the batch size to be 1 for all the platforms to achieve the minimum latency per image. The results of CPU and GPU are based on the original SSD300 algorithm using single-precision floating-point numbers, and the FPGA results are based on the hardware-friendly SSD300_HW as in Table 2, which uses 8-bit weights and 16-bit features with dynamic quantization to achieve the same accuracy level as software. Aided by

the customized hardware architecture specific for CNN inference acceleration, Arria 10 achieves 53× higher performance than CPU and Stratix 10 obtains 1.12× better throughput than GPU, even if FPGAs suffer from lower clock frequency and much less memory bandwidth. Due to the difficulty of directly measuring the power of CPU, GPU and FPGA, the listed power numbers are from their datasheet specifications for only rough estimation. Based on this, Arria 10 and Stratix 10 FPGAs can achieve 3.3× and 2.8× better energy-efficiency compared to GPU with 6.3× and 2.5× less power consumption, respectively.

7 CONCLUSIONS

In this work, we presented an efficient hardware implementation of the SSD300 object detection algorithm, tailored for an FPGA. The proposed design, SSD300_HW, achieves this through three basic innovations. These are: 1) replacing the dilated convolution with a normal convolution, 2) using a constant scale instead of normalization, and 3) using a uniform convolution sliding stride. Fixed-point arithmetic is employed to reduce the computation resource usage, which significantly enhances the FPGA inference performance, and the dynamic quantization is used to remain the detection accuracy of floating-point representation. The proposed FPGA-based inference engines achieve 1.03 TOPS and 2.18 TOPS throughput for SSD300_HW on Intel Arria 10 and Stratix 10 FPGA, respectively, and they also consume 6.3× and 2.5× less power and obtain 3.3× and 2.8× better energy efficiency, respectively, compared to a high-end GPU.

8 ACKNOWLEDGMENT

This work was supported in part by the NSF I/UCRC Center for Embedded Systems through NSF grants 1230401, 1237856, 1701241, 1361926 and 1535669, NSF grants 1652866 and 1715443, Intel Labs, and Samsung Advanced Institute of Technology.

REFERENCES

[1] Utku Aydonat, Shane O'Connell, Davor Capalija, Andrew C. Ling, and Gordon R. Chiu. 2017. An OpenCL™Deep Learning Accelerator on Arria 10. In *ACM/SIGDA Int. Sym. on Field-Programmable Gate Arrays (FPGA)*.

[2] M. Everingham, L. Van Gool, C. K. I. Williams, J. Winn, and A. Zisserman. [n. d.]. The PASCAL Visual Object Classes Challenge 2012 (VOC2012) Results.

[3] Yijin Guan, Hao Liang, Ningyi Xu, Wenqiang Wang, Shaoshuai Shi, Xi Chen, Guangyu Sun, Wei Zhang, and Jason Cong. 2017. FP-DNN: An Automated Framework for Mapping Deep Neural Networks onto FPGAs with RTL-HLS Hybrid Templates. In *IEEE Int. Sym. on Field-Programmable Custom Computing Machines (FCCM)*. 152–159.

[4] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Jun. 2016. Deep Residual Learning for Image Recognition. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*.

[5] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. 2014. Caffe: Convolutional Architecture for Fast Feature Embedding. *arXiv preprint arXiv:1408.5093* (2014).

[6] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. 2012. ImageNet Classification with Deep Convolutional Neural Networks. In *Advances in Neural Information Processing Systems (NIPS)*.

[7] Wei Liu, Dragomir Anguelov, Dumitru Erhan, Christian Szegedy, Scott E. Reed, Cheng-Yang Fu, and Alexander C. Berg. Oct. 2016. SSD: Single Shot MultiBox Detector. In *European Conference Computer Vision (ECCV)*.

[8] Yufei Ma, Yu Cao, Sarma B. K. Vrudhula, and Jae-sun Seo. 2017. An automatic RTL compiler for high-throughput FPGA implementation of diverse deep convolutional neural networks. In *Int. Conf. on Field Programmable Logic and Applications (FPL)*.

[9] Yufei Ma, Yu Cao, Sarma B. K. Vrudhula, and Jae-sun Seo. 2017. Optimizing Loop Operation and Dataflow in FPGA Acceleration of Deep Convolutional Neural Networks. In *ACM/SIGDA Int. Sym. on Field-Programmable Gate Arrays (FPGA)*.

[10] Bert Moons and Marian Verhelst. 2017. An Energy-Efficient Precision-Scalable ConvNet Processor in 40-nm CMOS. *J. Solid-State Circuits* (2017).

[11] Jiantao Qiu, Jie Wang, Song Yao, Kaiyuan Guo, Boxun Li, Erjin Zhou, Jincheng Yu, Tianqi Tang, Ningyi Xu, Sen Song, Yu Wang, and Huazhong Yang. 2016. Going Deeper with Embedded FPGA Platform for Convolutional Neural Network. In *ACM/SIGDA Int. Sym. on Field-Programmable Gate Arrays (FPGA)*.

[12] Joseph Redmon, Santosh Kumar Divvala, Ross B. Girshick, and Ali Farhadi. 2016. You Only Look Once: Unified, Real-Time Object Detection. In *IEEE Conf. on Computer Vision and Pattern Recognition (CVPR)*.

[13] Shaoqing Ren, Kaiming He, Ross B. Girshick, and Jian Sun. Dec. 2015. Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks. In *Advances in Neural Information Processing Systems (NIPS)*.

[14] Dongjoo Shin, Jinmook Lee, Jinsu Lee, and Hoi-Jun Yoo. 2017. 14.2 DNPU: An 8.1TOPS/W reconfigurable CNN-RNN processor for general-purpose deep neural networks. In *IEEE Int. Solid-State Circuits Conference (ISSCC)*.

[15] Karen Simonyan and Andrew Zisserman. 2014. Very Deep Convolutional Networks for Large-Scale Image Recognition. *CoRR* abs/1409.1556 (2014). arXiv:1409.1556 http://arxiv.org/abs/1409.1556

[16] Naveen Suda, Vikas Chandra, Ganesh Dasika, Abinash Mohanty, Yufei Ma, Sarma Vrudhula, Jae-sun Seo, and Yu Cao. 2016. Throughput-Optimized OpenCL-based FPGA Accelerator for Large-Scale Convolutional Neural Networks. In *ACM/SIGDA Int. Sym. on Field-Programmable Gate Arrays (FPGA)*.

[17] Xuechao Wei, Cody Hao Yu, Peng Zhang, Youxiang Chen, Yuxin Wang, Han Hu, Yun Liang, and Jason Cong. 2017. Automated Systolic Array Architecture Synthesis for High Throughput CNN Inference on FPGAs. In *Design Automation Conference (DAC)*.

[18] Fisher Yu and Vladlen Koltun. 2015. Multi-Scale Context Aggregation by Dilated Convolutions. *CoRR* abs/1511.07122 (2015). arXiv:1511.07122 http://arxiv.org/abs/1511.07122

[19] Chen Zhang, Zhenman Fang, Peipei Zhou, Peichen Pan, and Jason Cong. 2016. Caffeine: towards uniformed representation and acceleration for deep convolutional neural networks. In *Int. Conf. on Computer-Aided Design (ICCAD)*.

[20] Chen Zhang, Peng Li, Guangyu Sun, Yijin Guan, Bingjun Xiao, and Jason Cong. 2015. Optimizing FPGA-based Accelerator Design for Deep Convolutional Neural Networks. In *ACM/SIGDA Int. Sym. on Field-Programmable Gate Arrays (FPGA)*.

[21] Ruizhe Zhao, Xinyu Niu, Yajie Wu, Wayne Luk, and Qiang Liu. 2017. Optimizing CNN-Based Object Detection Algorithms on Embedded FPGA Platforms. In *Applied Reconfigurable Computing (ARC)*.