Energy-Efficient and High-Throughput CNN Inference Engine Based on Memory-Sharing and Data-Reusing for Edge Applications

Md. Najrul Islam, *Graduate Student Member, IEEE*, Rahul Shrestha, *Senior Member, IEEE*, and Shubhajit Roy Chowdhury, *Senior Member, IEEE*

***Abstract*— This paper proposes implementation-friendly and dynamically reconfigurable VLSI-algorithm for convolutional neural network (CNN) inference engine. Based on this algorithm and additionally suggested techniques, high-throughput and hardware-efficient architecture of kernel processing unit (KPU) for the CNN inference engine has been presented here. It specially enables the proposed CNN inference-engine to achieve efficient local data reuse for all the computations of state-of-the-art CNN models. Hardware implementation of such KPU on Zynq UltraScale+ ZCU102 FPGA-board is capable of delivering 3.68× higher throughput and 3.40× higher energy-efficiency than the contemporary designs in the literature. Furthermore, this work suggests hardware-efficient architecture of classify unit for the CNN inference engine. It delivers 79.44% better hardware efficiency than the state-of-the-art work, when implemented on FPGA platform. In addition, aforementioned efficient-architectures of KPU and classify unit are aggregated to construct energy-efficient and high-throughput design of a complete CNN inference-engine. It has been FPGA implemented, and ASIC synthesized as well as post-layout simulated in 28 nm FD-SOI technology node. Hence, the suggested CNN inference engine with 864 processing elements delivers a peak throughput of 6.65 TOPs while operating at a maximum clock frequency of 3.85 GHz. To the best of authors' knowledge, such unified design and implementation of CNN inference engine (including both KPU and classify unit), which is specifically capable of supporting efficient reuse of local data for all computation while processing the state-of-the-art CNN models, has been reported for the first time in this paper. Finally, the proposed CNN inference engine has been functionally validated in the real-world test scenario for the object classification application, using contemporary CNN models.**

***Index Terms*— Convolutional neural network (CNN), very large scale integration (VLSI), digital architecture design, field programmable gate array (FPGA), fully depleted silicon on insulator (FD-SOI), application specific integrated circuit (ASIC).**

I. Introduction

SUPERIOR accuracy of CNN models has spiked the development of modern artificial-intelligence (AI) applications like object detection, audio classification, natural language processing, health care monitoring, financial fraud detection, autonomous driving and many more [1], [2], [3], [4]. However, the requirements of sophisticated and massive computations that consume huge power for CNN models incur severe bottleneck in the widespread deployment of such applications on edge devices [5]. This necessitates the development of high-throughput and energy-efficient processing units for the implementation of such CNN applications. By using multiple processing elements (PEs) in parallel, CNN inference engines for these AI applications can achieve remarkable surge in speed. However, at the same time, their energy consumption increases which is still a major concern from an implementation aspect. Recent study [6], [7] has shown that significant portion of total power consumption in CNN computation is largely contributed due to frequent and massive movement of data between off-chip dynamic random-access-memory (DRAM) and processing unit. Such power dissipation is rather more than the power required by the computational processing units. Furthermore, the parallel processing using multiple PEs is restricted due to limited bandwidth of off-chip DRAM [8], [9], [10]. As a result, PEs in large array frequently starve for the data that often causes interruption in their processing and hence, limits the capability of sustaining peak throughput of CNN inference engine [9], [11].

Several works in the literature have proposed hardware architectures to perform computation for CNN models with higher throughput and better energy-efficiency [6], [7], [12]. These reported implementations mostly exploited the reuse of local data, at the expense of extra hardware requirements for complex routing and additional memory blocks. Such designs mainly focused on the architectural optimization for the computation of convolutional operation (denoted as *Conv*) in feature extraction layers. However, as energy consumption is significantly contributed by data movements, optimizing both processing and data-movement of other computations in the feature extraction layer, like *ReLU*, *MaxPool*, and *AvgPool*, are equally important. Thus, it is necessary to minimize the data movement for the computation of all layers in CNN model to scale-up the energy efficiency of any CNN-based applications. Therefore, this work caters such needs by proposing reconfigurable PE-architecture that performs all types of computations in the feature extraction layers for the state-of-the-art CNN models, that enables to reuse the local data for all the computations of feature extraction layers. On the other hand, conventional energy-efficient

Manuscript received 27 October 2023; revised 10 March 2024 and 16 April 2024; accepted 20 April 2024. Date of publication 7 May 2024; date of current version 28 June 2024. This article was recommended by Associate Editor Y. Zhang. *(Corresponding author: Rahul Shrestha.)*

The authors are with the School of Computing and Electrical Engineering, Indian Institute of Technology (IIT) Mandi, Suran 175005, India (e-mail: d18064@students.iitmandi.ac.in; rahul_shrestha@iitmandi.ac.in; src@iitmandi.ac.in).

Color versions of one or more figures in this article are available at https://doi.org/10.1109/TCSI.2024.3392807.

Digital Object Identifier 10.1109/TCSI.2024.3392807

architecture of CNN inference engine like [7], [8], [9], and [10] utilizes large amount of hardware for memory as well as routing-network to reuse the local data and such inference engine often encounters interruptions in their computation which is periodically halted during data movement in the PE array. This further affects the sustainable throughput of CNN inference engine. To circumvent such problems, our work presents a shared line-memory based PE array by using a hardware-efficient line memory to achieve the reuse of local data and uninterrupted processing by utilizing significantly lesser memory as well as other hardware resources.

On the other hand, conventional softmax-based classification layer [13], [14] apparently requires N number of complex exponential and the same number of divisive operations to compute the probability of each class for an N-class model. Thus, while majority of CNN inference engines [14] rely on off-chip processor for the computation of classification layer, various attempts have been made to accelerate the computation of classification layer by using dedicated hardware. However, these works use coordinate-rotations digital-computer (CORDIC) based method [15] or look-up table (LUT)-based powers-2 approximation [13], [14], [16], [17], [18] to realize exponential and divisive computations. Unfortunately, such implementations still require significant amount of hardware resources, and these hardware designs are not integrated with CNN inference engine to deliver complete on-chip solution [19]. Based on our analysis towards the working of classification layer and empirical realizations, we propose a simplified classification approach and hardware efficient classify unit where the computation of classification layer has been carried out using only comparators and multiplexers. Thus, this design excludes the requirement of any complex hardware for exponential and divisive computations. In addition, the suggested classify unit has been integrated with the kernel processing unit of CNN inference engine to build a complete on-chip engine. Highlights of all the contributions presented in this paper are enumerated as follows.

- A dynamically reconfigurable hardware-friendly and high-speed VLSI-algorithm for the complete CNN inference-engine has been proposed in this work. It especially includes memory-efficient uninterrupted processing and all computations of feature-extraction as well as classification layers.
- A new multi-purpose hardware-architecture for PE has been suggested here that is capable of performing all types of computations for the feature extraction layers of the state-of-the-art CNN models. In addition, this work also presents hardware-efficient architecture of the line memory for achieving efficient data-reuse and uninterrupted processing in CNN inference engine.
- Using the aforementioned PE and line-memory architectures, we have presented high-throughput and energy-efficient architecture of kernel processing unit. Subsequently, hardware efficient design of classify unit has been suggested here that is integrated with kernel processing unit to realize complete design of efficient CNN inference-engine.
- Comprehensive analyses of accuracy (using standard data-sets) and hardware resources for both the proposed algorithm and architectures, respectively, are carried out. Finally, we presented FPGA implementation of our CNN inference engine and its functional validation, using real-world test setup. It is also ASIC synthesized and post-layout simulated in 28 nm FD-SOI technology node.

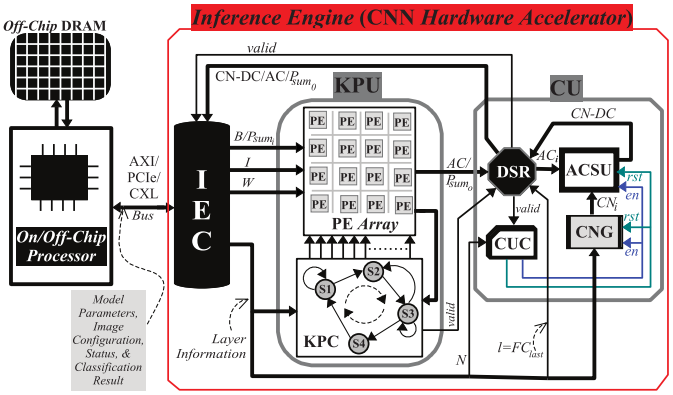

Fig. 1. Overall architecture of an objection classification system based on the proposed CNN inference engine.

II. System Model and Proposed Algorithm for Inference Engine

A. System Model

A generic overview of the proposed object-classification system is shown in Fig. 1, consisting of three major parts: an on/off-chip processor, an off-chip DRAM, and a CNN inference engine. Primary role of on/off-chip processor is to handle the software-based tasks like loading model parameters from off-chip non-volatile memory, input the image data from external source like camera as well as store them using off-chip DRAM, and provide configuration signals to the inference engine, as presented in Fig. 1. It also shows that inference engine and on/off-chip processor are interfaced via high data-rate links like CXL or PCIe or AXI buses for transferring data between off-chip DRAM and inference engine. The key role of inference engine is to perform dedicated high-speed computations for the CNN model. Since such computation types and processing requirements for different layers of CNN model vary over a wide range, the inference engine has been designed with three major blocks: kernel processing unit (KPU), classify unit (CU), and inference engine controller (IEC), as shown in Fig. 1. Here, IEC communicates with on/off-chip processor, KPU, and CU, in order to configure and manage the flow of data like input feature map (I), filter weights (W), partial sum (P_{sum_i}), bias values (B), output activation values (AC), classification result (CN-DC) and many more signals. Furthermore, IEC is responsible for managing the data flow between inference engine and off-chip DRAM, as illustrated in Fig. 1.

To begin with, IEC first configures both KPU and CU with the layer information of CNN model, and also provides input image as I along with W as well as B of the active layer of model to KPU. Depending on the size of PE array in KPU and parameter size of the current layer, KPU either generates the final AC value of the layer or only a portion of AC, referred as partial sum (P_{sum_o}). These partial sums are used in

successive iterations of computation to generate the final AC and eventually, output from KPU is fed to CU. If the current layer (l) is last fully connected layer (i.e. $l{=}FC_{last}$) of the CNN model then CU starts the computation of classification layer. Finally, either CU returns the class no. (CN) of the object detected in the input image back to IEC or CU directly returns the output AC from KPU back to IEC in the form of partial sum (P_{sum_o}) or AC, as shown in Fig. 1. Here, IEC reuses these values during successive iterations of computation for the same or next layer. Comprehensive discussion on the proposed efficient architectures of KPU and CU are presented in Sections III-A and III-B, respectively.

B. Proposed Algorithm

The proposed implementation-friendly algorithm for complete CNN inference-engine has been presented in Algorithm 1. Its salient features are to dynamically configure the inference engine based on the layer information, to uninterruptedly perform the computations, and to classify the object in CNN inference engine itself while processing the last FC layer. To begin with, values must be provided for three input parameters: (1) layer number of the last FC layer (FC_{last}), (2) number of iterations required for a layer (n_l), and (3) minimum number of data items that are required to begin the processing of that layer (r_l), as presented in line nos. 2-3 of Algorithm 1. Subsequently, selection of computation type that is required by the layer is performed in line no. 4 of Algorithm 1. As discussed earlier, W, I, B, and AC denote filter weight, input feature map, bias value, and output activation, respectively. In addition, a, b, c, and d represent the current position(s) in one, two, three, and four dimensional spaces of the data, respectively, that is being processed during current stride. Here, $\alpha{\times}\beta{\times}\gamma{\times}\delta$ is the size of a multi-dimensional filter, referring line no. 4 in Algorithm 1, where width, height and depth of a filter are represented by α, β and γ, respectively. Further, δ denotes the number of filters.

To achieve uninterrupted processing, the pre-fetching (or partial fetching) of r_l data is first completed and thereafter, the computation begins simultaneously with the fetching of remaining data, as presented in line nos. 5-15 of Algorithm 1. Once the data loading is over for the current iteration, pre-fetching of data commences for the subsequent iteration, even in the case of layer switching, as shown in line nos. 14-23 of Algorithm 1. After the processing completes for iteration, next iteration immediately starts. Here, if the processing of current layer (in a given iteration) is the last FC layer (i.e. $l{=}FC_{last}$) then the computation for classification layer is performed in pipeline where the activation (AC_i) is fed as input and eventually, returns the class no. (CN) of the detected class as an output, referring line nos. 24-28 from Algorithm 1. On the other side, if $l{\neq}FC_{last}$ then AC_i is returned as output that is recycled during the computation of subsequent layer, as illustrated by line no. 28 of Algorithm 1.

To perform the computation of average pooling layer, the denominator of division has been replace from $\alpha.\beta$ to $2^{\log_2^{\lfloor\alpha.\beta\rfloor}}$ that enables to perform the division operation using simple bit shifting, rather than using complex hardware for division operation. Since all values of the activation are scaled at same ratio, no degradation has been noticed in the classification accuracy of the model. Furthermore, for the computation of classification layer, our algorithm employs a simplified classification technique wherein it generates the classification result directly from the activation of last FC layer, and avoids all complex exponential and divisive computations that are used in conventional softmax-based classification layer processor. This new approach is based on our analysis towards the working of classification layer and empirical realizations. As a result, it is

Algorithm 1 Proposed Implementation-Friendly Algorithm for Inference Engine

1: **Initialization:** $i{=}1$, $j{=}0$, $l{=}0$, $AC\text{-}Max{=}0$, & $CN\text{-}DC{=}0$; ▷ i, j, and l count the number of processed iteration, the number of data that has been fetched, and the number of processed layer, respectively.
2: Determine FC_{last}; ▷ FC_{last} is layer number of the last FC layer.
3: Determine n_l and r_l; ▷ n_l, and r_l are number of iterations required for $Layer_l$ & minimum number of data required to begin the computation of $Layer_l$.
4: Determine Computation (P^l) as:

$$P^l \Rightarrow AC = \begin{cases} \sum_{a}^{+\alpha}\sum_{b}^{+\beta}\sum_{c}^{+\gamma}\left(W_{a,b,c,\delta}\times I_{a,b,c}\right)+B_d, & Layer_l = Conv \\ \sum_{a=1}^{n} I_a \times W_a + B, & Layer_l = FC \\ Max\left\{I(a{:}{+}\alpha, b{:}{+}\beta,\ c)\right\}, & Layer_l = MaxPool \\ \dfrac{\sum_{a}^{+\alpha}\sum_{b}^{+\beta} I_{a,b,c}}{2^{Log_2^{\lfloor\alpha.\beta\rfloor}}}, & Layer_l = AvgPool \\ Max\{0, I_a\}, & Layer_l = ReLU \\ Min\{Max\{0, I_a\}, 6\}, & Layer_l = ReLU6 \end{cases}$$

5: Begin $F^l_{i=1}$; ▷ Start pre-fetching of data for $itr^l_{i=1}$.
6: **while** $i \leq n_l$ **do**
7: **if** $j \leq r_l$ **then** ▷ Enough data has not been fetched yet.
8: Continue $F^l_{i=1}$; ▷ Fetching continues for $itr^l_{i=1}$.
9: $j = j{+}1$; ▷ Count the number of data pre-fetched for $itr^l_{i=1}$.
10: Return to Step 7; ▷ Fetching continues for $itr^l_{i=1}$.
11: **else** ▷ Enough Data Fetched to Begin Computation.
12: **Process** P^l_i; ▷ Compute AC_i.
13: **if** F^l_i is not complete **then**
14: Continue F^l_i;
15: **else**
16: **if** $i < n_l$ **then**
17: Begin F^l_{i+1}; ▷ Start pre-fetching data for itr^l_{i+1}.
18: **else**
19: set j = 0
20: Begin $F^{l+1}_{i=0}$; ▷ Start pre-fetching data for $itr^{l+1}_{i=0}$.
21: **end if**
22: **end if**
23: **if** P^l_i is Complete **then**
24: **if** $l = FC_{last}$ **then**
25: $$AC_{Max_i} = \begin{cases} AC_i, & AC_i > AC_{Max_{i-1}}; \\ AC_{Max_{i-1}}, & AC_i \leq AC_{Max_{i-1}}; \end{cases}$$
26: $$CN\text{-}DC_i = \begin{cases} CN\text{-}DC_{i-1}, & AC_i < AC_{max_{i-1}}; \\ CN_i, & AC_i \geq AC_{Max_{i-1}}; \end{cases};$$
27: **end if**
28: $$Output = \begin{cases} AC_i, & l \neq FC_{last}; \\ 0, & l = FC_{last}\ \&\ i \neq n_l; \\ CN\text{-}DC_i, & l = FC_{last}\ \&\ i{=}n_l; \end{cases}$$
29: $$l = \begin{cases} l+1, & i{=}n_l\ \&\ l \neq FC_{last}; \\ 0, & i{=}n_l\ \&\ l = FC_{last}; \\ l, & i < n_l; \end{cases}$$
30: **if** $i = n_l$ **then**
31: Set $i = 0$;
32: Return to Step 3. ▷ Start processing the next layer.
33: **else**
34: Set $i = i + 1$
35: Return to Step 12. ▷ Start computation for itr^l_{i+1}.
36: **end if**
37: **else**
38: Return to Step 12 ▷ Continue P^l_i.
39: **end if**
40: **end if**
41: **end while**

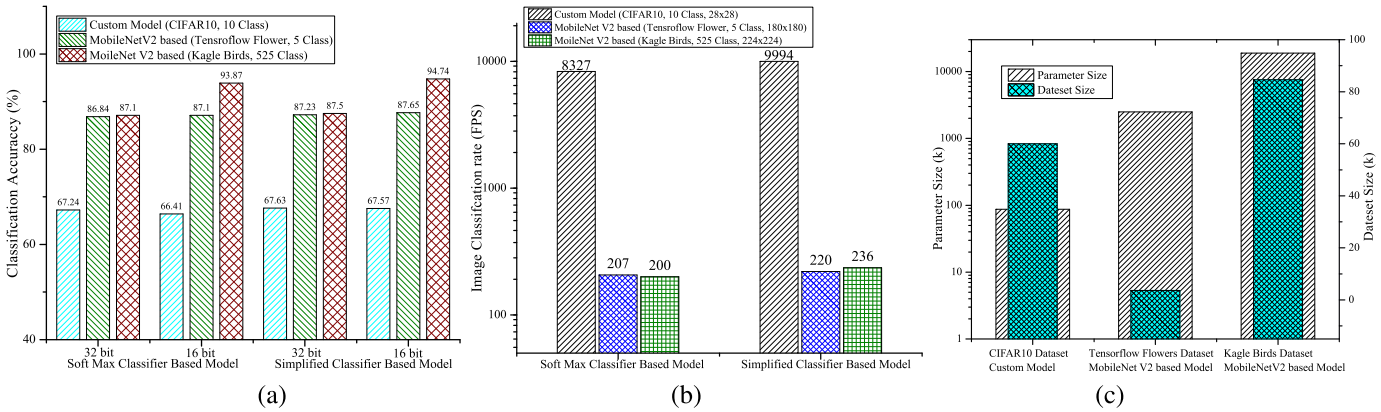

Fig. 2. Comparative accuracy and performance analyses of the proposed implementation-friendly algorithm for CNN inference engine.

found that the probability (P_i) of an object class is maximum when the magnitude of activation AC_i, associated with that class from the last FC layer, is at its peak. This method processes N (i.e. $N{=}n_{FC_{last}}$) number of AC_i and it subsequently searches as well as determines the largest activation (AC_{Max}) and its associated class no. (CN), as presented in line nos. 25-29 of Algorithm 1. Finally, it returns the class number associated with AC_{Max} as the detected class, referring line no. 26-28 in Algorithm 1. Aforementioned proposed technique uses simple comparison and sorting method. Thereby, its corresponding hardware architecture (i.e. classify unit (CU) from Fig. 1) requires only comparator and multiplexer.

Comparative performance analysis of both the proposed and the conventional techniques are presented in Fig. 2. It shows the comparison of classification accuracy in Fig. 2 (a), and computation frame rate in Fig. 2 (b) of three different CNN models for both conventional and proposed approaches. Here, three CNN models have been developed for three different datasets. Furthermore, Fig. 2 (c) shows the size of dataset used for training these models and their parameter size. Referring Fig. 2, unlike parameter and dataset size, choosing proposed hardware-friendly approach over the conventional approach refrains from causing noticeable impact on the classification accuracy. Moreover, it significantly improves the processing speed and the classification frame rate.

III. Proposed Hardware Architectures, Implementation Results and Comparisons

Based on the proposed Algorithm 1, this section presents efficient hardware architectures of kernel processing unit (KPU) and classify unit (CU) for the complete CNN inference-engine, as discussed earlier in Section II with the aid of Fig. 1. In addition, their implementation results are also presented and compared with the state-of-the-art works.

A. Energy and Memory Efficient Architecture of Kernel Processing Unit (KPU)

As discussed earlier in Section I, an energy-efficient and high-speed inference engine must have the capability to minimize the costly off-chip memory access by maximizing the efficient reuse of local data within the chip to alleviate power consumption and mitigate interruptions in the data processing to sustain the peak achievable throughput. For such processing, the proposed KPU architecture has been designed based on the line stationary approach, as shown in Fig. 3. It uses a specially designed line memory to store I data within the PE array, and W value inside the PE. As shown in Fig. 3, the suggested KPU consists of $m{\times}n$ PE matrix wherein m line-memory units are used for three key purposes: (1) to store the incoming data, (2) for providing data to PEs, and (3) to reuse the data within the PE array. In order to efficiently configure the PE array, for different filter shapes, and achieve efficient data-reuse, the routing of data between line memories and PEs has been controlled with the aid of kernel processing controller (KPC), as presented in Fig. 3. At a given time instance, any PE in the array can be dynamically routed to and fetch the data from any of the m line memories in our design. At a time, one line memory can provide data to n different PEs. Thus, suggested KPU architecture that contains PE array and line memories can perform multiply-&-accumulate (MAC) and pool operations for varying filter sizes, ranging from $1{\times}1$ to $m{\times}n$, at different level of parallelism. Comprehensive discussion on the proposed architectures of PE and line memory are carried out in the following Sections III-A.1 and III-A.2, respectively. Furthermore, detailed explanation of the techniques which allow the proposed KPU to achieve energy-efficient data reuse and uninterrupted processing are presented in Sections III-A.3 and III-A.4, respectively.

1) Multipurpose PE Architecture: Referring line no. 4 of Algorithm 1, the inference engine must dynamically configure itself for different types of computations viz. *Conv*, *FC*, *MaxPool*, *AvgPool*, *ReLu*, and *ReLu6*, which are required by various layers of the CNN model. Therefore, KPU of CNN inference engine requires a PE which supports all these computation types. Hence, Fig. 4 shows the proposed micro-architecture of such PE that consists of MAC unit for processing both convolutional (i.e. *Conv*) and fully connected (i.e. *FC*) layers; a MAX module that performs the max pooling operation (i.e. *MaxPool*). As illustrated in Fig. 4, depending upon the value of $MAC/\overline{MAX}$ select signal, one of either MAC or *MaxPool* operation is selected. In order to perform *ReLu* activation on I_l (i.e. A_{l-1}) as well as to conserve energy for null values, the suggested PE also consists of sign-and-zero detector (SZD). On detecting a negative or null value of input data (I), SZD turns off the MAC unit and produces a null

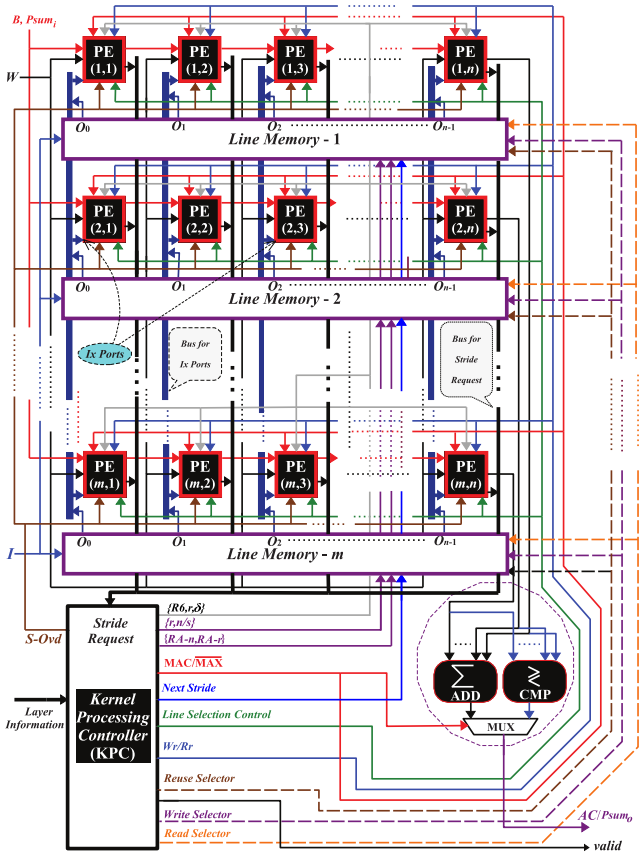

Fig. 3. Proposed hardware architecture of energy and memory efficient KPU.

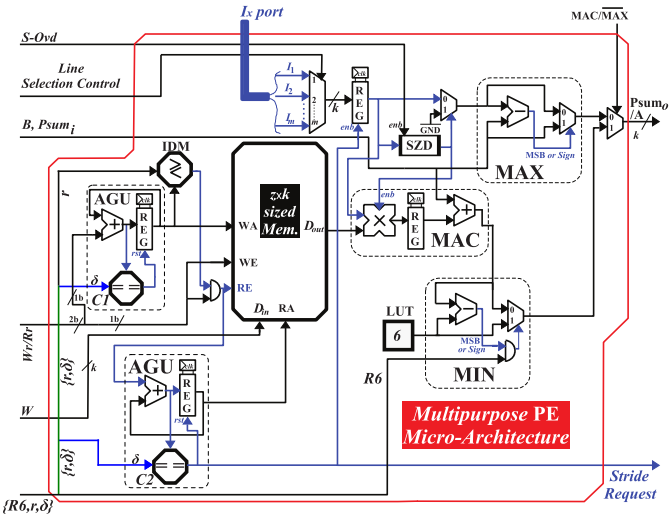

Fig. 4. Proposed architecture of multipurpose PE that is used in the design of PE-array for KPU.

value at the output. In this way, it performs *ReLu* activation on A_{l-1} and conserves energy by avoiding unnecessary switching of MAC unit for such input values. This plays a significant role in energy conservation, as the major portion of I values are zeros towards the later layers of CNN models. Moreover, SZD is overridden by the *S-Ovd* signal for first layer of the model to allow I_1, irrespective of their polarity. Furthermore, to perform *Relu6* on the outgoing activation, a MIN module is used which clips the output activation below 6 when required. To perform average pooling (*AvgPool*) operation, multiplier in the MAC unit acts as a pass-through buffer, thus MAC unit performs as an adder only. The accumulated results are bit shifted to achieve the division without using a real hardware for divider (referring line no. 4 of Algorithm 1). As discussed in Section III-A, our PE can fetch data from any of the m different line memories. Data from the desired line memory is selected by using the line-selection control signal, as shown in Fig. 4.

Since CNN models apply several filters on the same input data, the proposed PE architecture also incorporates a $z\times k$-sized memory, which stores z filter weights of k-bit each, in order to minimize the off-chip data access and maximize the local data reuse. Here, read and write operations are enabled with the aid of two independent address-generation units (AGUs) for read and write address ports: RA and WA ports, as shown in Fig. 4. However, read operation is supervised by the input data monitor (IDM) module which disables the read operation until r number of filter weights are not stored in the memory. Here, each of the I values that is fed to PE is reused upto $z\times$ by z different filter weights. As a result, rather than fetching a new I data in each clock cycle, the proposed PE uses stride request based mechanism. Once in every δ clock cycles when AGU reads the last address of the filter memory, comparator *C2* in the proposed PE architecture sends the stride request signal (i.e. *Stride Request*) to KPC module for the next I data, as well as activates input data register to store the incoming data and holds it for next δ clock cycles, as shown in Fig. 3 and 4.

2) Hardware-Efficient Line-Memory Architecture: Conventional energy-efficient designs of inference engine [6], [7] use large memories inside the PE to store filter weights, input data as well as partial sums, and continuously move these data in all directions for their local reuse. Such implementation requires significant hardware and energy overheads for storage and routing purposes. It also faces frequent interruptions in the processing while moving the data in and out of PEs. On the other hand, as discussed in Section III-A, our KPU architecture uses the line stationary approach that simultaneously shares one single line memory with n different PEs, as presented in Fig. 3.

The proposed architecture of a line memory is shown in Fig. 5. It comprises of $k\times A$-sized memory which is adequate for storing at least A number of input data where A represents the size of a single row of the input feature map I. Here, read and write operations in such line memory are independently governed by two different control signals from the KPC module: read selector and write selector, as shown in Fig. 5. These signals decide whether the line memory stores the incoming data or provides the data to PEs. In addition, they locally manage read and write addresses of the memory by controlling two different address generator units: write address generator (WAG) and read address generator (RAG), as shown in Fig. 5. Furthermore, read operation is supervised by IDM which ensures such operation is being performed only when the adequate number of data (i.e. r) has been fetched from the memory (referring line nos. 7-13 of Algorithm 1). Whenever the write-selector signal selects a line memory for writing

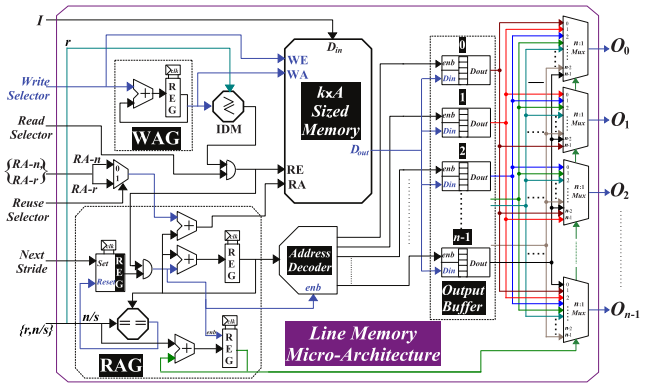

Fig. 5. Proposed architecture of single line memory, used in the design of PE array for KPU.

the incoming data, WAG sequentially generates a new write address in every clock cycle to store each of the incoming data items. During every stride, line memory receives two different read pointers (*RdPtr*): RA-r and RA-n from the KPC module of KPU to denote reading locations for reused and new data, respectively. Hence, if the data stored in the line memory is being reused during a vertical stride then it uses RA-r (to indicate read address for reuse) as the read pointer, that follows the same addressing scheme which was adopted in the previous vertical stride. Otherwise, it uses RA-n as the read pointer if the data of line memory is being used for the first time. Here, the reuse selector signal from KPC module has been generated to indicate, if a line memory is selected for reused or fresh data. When a line memory is selected for read operation and if IDM indicates that sufficient data has been written into the memory, then every time RAG receives a *Next Stride* trigger-signal, it generates n/s consecutive read addresses that starts from the address pointed by *RdPtr*, over the next n/s clock cycles.

In addition, line memory has an output buffer that comprises of n k-bit registers and the same number of n:1 multiplexers, as shown in Fig. 5. Apart from generating read addresses for the $k\times A$ memory, RAG also generates write address for the registers of this output buffer, and select line bits for the multiplexers. When read operation is being performed in the line memory, during each clock cycle, address decoder module decodes these write addresses and activates one of the n registers to store the output of $k\times A$ memory, as shown in Fig. 5. In this way, every time there is a trigger in the *Next Stride* signal, n/s number of new data items from n/s different locations of $k\times A$ memory are buffered in n registers of the output buffer. Thus, between every two horizontal strides, output buffer holds $n-s$ number of old data for re-use and s number of new data. Here, select-line bits for the multiplexer also increments by s to switch connection of output ports. As discussed earlier in Section III-A and shown in Fig. 3 as well as Fig. 4, n parallel outputs (O_0 to O_{n-1}) from n registers of the output buffer of a single line memory is connected to any one of the m different Ix ports of n different PEs. Note that Ix ports is represented as I_1, I_2, I_3 $\cdots$ I_m (each of k bit-width) which are fed to m:1 multiplexer in the proposed PE architecture in Fig. 4. Here, single outputs from m line-memories are connected to every PE via such Ix port, as shown in Fig. 3. Referring Fig. 4, every time there is a stride request (once in every δ clock cycles), data from O port of line memory is transferred to the input register of the PE (via Ix port and m:1 multiplexer of PE architecture). Thus, registers of output buffer in the line memory can again store the data which are required in the next stride. Therefore, the proposed line-memory architecture uses only single $k\times A$-sized memory for n number of PEs, compared to n number of such memories in conventional architectures from [7].

3) Technique for Efficient Data Reuse: As discussed earlier in Section I, power requirement and energy efficiency of KPU mainly depends on two factors: (1) the way KPU handles off-chip data movement, and (2) how efficiently does KPU locally reuses data within the chip? To reduce the number of off-chip memory accesses, each of I values that is being fed to each PE in the proposed KPU architecture can be reused $z\times$, as discussed in Section III-A.1. In addition, every time the *Next Stride* signal from line memory is triggered high to perform horizontal strides, the KPC module in KPU increments read pointer (*RdPtr*) by s which denotes the stride size. Thus, horizontal strides are performed simply by providing I data to all n PEs from a new location of the same line memory. As a result, between every two consecutive vertical strides, each data is used by $\alpha-s$ number of PEs for the same number of horizontal strides. Therefore, between every two consecutive vertical strides, data values are reused up to $z(\alpha-s)\times$ in our design. As discussed earlier in Section III-A, at a given time instance, any PE in the array can be provided with the data from any of m line memories that is illustrated in Fig. 3. Here, the vertical strides in PE array is managed by the line-selection control signal from the KPC module that switches the connection of a PE from x^{th} line memory to $(x+s)^{th}$ line memory to perform a vertical stride with a stride size of s. For the better visualization of this technique, 3×3 PE array has been considered with three line memories and a stride size $s=1$ on an input feature map I of size $A\times A$, as presented by Fig. 6. It shows the process of reusing the data during vertical stride, and pre-fetching of data to perform uninterrupted computation. Basically, the process of vertical stride has been carried out in such a way that the top $\alpha-s$ rows of PEs are routed to line memories which were connected to their subsequent PE rows, during the previous vertical stride, as presented in Fig. 6. Furthermore, bottom s rows of PE are routed to line memories where data has been fetched for the current vertical stride.

Referring Fig. 6 (a), three line memories viz. Line Memory-1, Line Memory-2 and Line Memory-3 are holding row-1, row-2, and row-3, respectively, of the input feature map I. Since the row-1 data fetched from Line Memory-1 is being used for the last time during the current vertical stride, initial r number of data in Line Memory-1 is not required anymore once the architecture performs r number of horizontal strides. Thus, this architecture pre-fetches initial r number of data item from row-4 of I to those initial r locations of Line Memory-1 (referring line nos. 16–22 from Algorithm 1). In second vertical stride, now the computation is to be performed on row-2, row-3, and row-4 of I. Since row-2 and row-3 are already there in Line Memory-2 and Line Memory-3, and

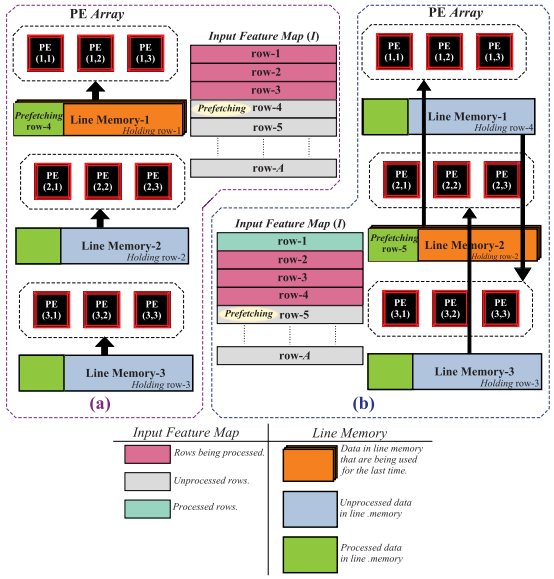

Fig. 6. Schematic representation of the proposed technique for efficient data reuse in PE array of KPU architecture.

also enough r number of data from row-4 is pre-fetched in Line Memory-1, computation for second vertical stride can immediately start after the computation of first vertical stride. Furthermore, Fig. 6 (b) shows the suggested process of reusing the data of Line Memory-2 and Line Memory-3 by row-1 and row-2, and row-3 using new data of row-4 of I. Here, line-selection control signal of PEs in row-1, row-2 and row-3 change their connections from I_1, I_2, and I_3 to I_2, I_3, and I_1, respectively, as shown in Fig. 6 (b). Thus, during each vertical stride, our PE array reuses the data of $\alpha-s$ line memories and fetches data from external memory to other s line memories. Therefore, data stored in a line memory is reused for $\alpha-s$ number of vertical strides. As discussed above, between every two consecutive vertical strides, data values are reused up to $z(\alpha-s)\times$. Thus, each of the data items stored in line memory has been used by up-to $z(\alpha-s)^2$ number of MAC operations, and each of the filter values is used for $(A-s)^2$ number of MAC operations.

4) Technique for High-Throughput Computation: Computation in the conventional inference engine stops while moving the data within an array of PEs and when the data are loaded from external memory [6], [7]. Since the data movements in inference engine takes place as frequently as the computations, achievable peak throughput of accelerator is adversely affected due to such frequent interruptions in the computation process that are mitigated by the proposed technique, as presented in Algorithm-1. Referring line nos. 7–13 in this Algorithm-1, instead of waiting for all the data for a vertical stride to be fetched in the line memory, suggested technique starts the computation after a partial fetching where a minimum r number of data are fetched and the remaining data will be fetched during the computation run time, referring line nos. 14–15 from Algorithm-1. This process is realized in the hardware architecture of KPU with the aid of suggested architectures of PE and line memory, as presented in Section III-A.1 and III-A.2. Here, the proposed design of PE requires new data only once in every δ clock cycles, and our line memory needs s clock cycles to generate the new data for s number of PEs. Therefore, it uses the remaining duration of $\delta-s$ clock cycles to fetch the remaining data from external memory. Thus, the proposed KPU performs both data fetching and computation, simultaneously. Furthermore, as discussed in Section III-A.3, suggested KPU architecture also pre-fetches the minimum number of data, required for subsequent vertical stride, and it immediately starts the computation of next vertical stride at the end of current vertical stride, without any interruption. Therefore, the proposed KPU architecture performs computation at highest throughput and sustains this peak throughput during the entire computation period.

5) PE Efficiency for Different Filter Sizes: The PE efficiency (Ω) in an inference engine is the ratio of number of utilized PEs and N_{PE}. As we know that the computational load in the state-of-the-art CNN models are mainly dominated by smaller-sized kernels, mostly 1×1 and 3×3 kernels. Thus, rather than large kernels with the size like 7×7 or 11×11 [11], our work proposed to re-organize $m\times n$ PE-array in a unique way to deliver 100% of PE efficiency (i.e. Ω=1) for most commonly used kernels of sizes 1×1 and 3×3. However, for 5×5 and 7×7 sized kernels, Ω of inference engine drops to 92% (i.e. Ω=0.92) and 91% (i.e. Ω=0.91), respectively. Therefore, an overall PE efficiency of the proposed KPU architecture is 100% (i.e. Ω=1) for widely-used CNN models like VGG as well as state-of-the-art models like MobileNet and EfficientNet, and it is $\approx$ 99% (i.e. Ω=0.99) for the models like GoogLeNet and ResNet.

6) KPU Implementation Results and Comparison: The proposed KPU architecture has been hardware implemented on FPGA platform (using AMD-Xilinx Zynq-Ultrascale+ ZCU102 MPSoC board). Detail implementation results of the proposed KPU are presented in Table I. For fair comparison, our KPU architecture has been additionally implemented in other FPGA platforms (viz. ZC706, VC707 and VC709) which are adopted by the reported state-of-the-art implementations. Note that the fixed-point bit-quantization representation ($Q_{n.m}$) of 16 bit has been used in all the aforementioned implementations. As we know, the computation throughput (Θ_T) determines overall processing performance of an inference engine because Θ_T is directly proportional to the inference rate. Such computation throughput is computed as $\Theta_T = 2(N_{PE}\times f_{clk}\times\Omega\times\sigma)$ number of operations per second where N_{PE} denotes the total number of PEs available in the KPU, f_{clk} is operating clock frequency of KPU. In addition, time efficiency σ is the ratio of time duration when PEs are active and total time duration required for the computation. Since in every clock cycle, a PE performs two distinct operations: one multiplication and one addition, thus a factor of 2 has been considered in the above expression for Θ_T to calculate the effective number of operations per second (OPs).

Static timing analysis of the proposed KPU architecture indicates that the critical path of this design lies in the PE and such path includes a single multiplier. Thus, our KPU implementation operates at a peak clock frequency of 1.25 GHz when implemented in aforementioned AMD-Xilinx Zynq-Ultrascale+ ZCU102 MPSoC FPGA-board. As shown

TABLE I
COMPARISON OF PROPOSED KPU IMPLEMENTATIONS WITH RELEVANT STATE-OF-THE-ART WORKS

	[20]	[22]	[23]	[21]	[12]	[11]	**Proposed Implementations**						
Platform	ZC706	ZC706	VC707	VC709	VC709	ZCU102	**ZC706**	**VC707**	**VC709**		**ZCU102**		
Clock Frequency (MHz)	200	150	200	200	200	340	**389**	**407**	**382**	**382**	**1250**	**1250**	**1250**
LUTS (in kilo)	–	182.616	–	121.472	107.325	103.985	**93.683**	**93.683**	**93.683**	**93.683**	**96.676**	**96.676**	**96.676**
FFs (in kilo)	–	127.653	–	159.872	74.430	20.631	**27.666**	**27.666**	**27.666**	**27.666**	**27.725**	**27.725**	**27.725**
BRAMs (36kb)	–	486	–	467	390	640	**64**	**64**	**64**	**64**	**64**	**64**	**64**
DSP Usage	576	780	64	664	0	864	**864**	**864**	**864**	**864**	**864**	**864**	**864**
N_{PE}	576	780	64	664	1312	864	**864**	**864**	**864**	**864**	**864**	**864**	**864**
Precision (in bits)	16/8	16	16	16	4	16	**16**	**16**	**16**	**16**	**16**	**16**	**16**
CNN Model	A.N*	VGG16	VGG16	VGG16	M.N.V2	VGG16	**VGG16**	**VGG16**	**VGG16**	**M.N.V2**	**VGG16**	**M.N.V2**	**R.N.50**
Θ_T (GOPs)	198.1	187.8	12.5	230.1	413.9	587.52	**672.192**	**703.23**	**660.01**	**660.01**	**2160**	**2160**	**2150**
η_{PE} (MOPs/PE)	343	241	195	347	320	680	**778**	**814**	**764**	**764**	**2500**	**2500**	**2475**
Frame Rate (fps)	–	6.12	–	–	419.3	18.9	**21.7**	**22.7**	**21.3**	**1073.2**	**69.7**	**3512.2**	**521.74**
Power Consumption (W)	–	–	–	9.61	6.10	4.11	**2.169**	**1.968**	**1.91**	**1.91**	**4.525**	**4.525**	**4.525**
Energy Efficiency (GOPs/W)	–	14.22	–	22.9	67.85	140.31	**309.91**	**357.33**	**345.55**	**345.55**	**477.35**	**477.35**	**475.14**

A.N* = Modified AlexNet, M.N.V2 = MobileNet V2, R.N.50 = ResNet 50

in Table I, suggested KPU can operates at 1.79×, 1.91×, 1.91×, and 3.68× higher clock frequency compared to the reported implementations from [11], [12], [20], and [21], respectively, when implemented in their respective FPGA boards. As discussed earlier in Section III-A.4, the proposed KPU architecture performs uninterrupted computations. Thus, PEs in our KPU architecture are never idle and carries out their computations with 100% of time efficiency (i.e. σ=1). Thus, the proposed KPU architecture is capable of delivering high throughput that is 3.39×, 2.87×, 1.6×, and 3.68× higher than the throughput achieved by [11], [12], [20], and [21] and reported implementations, respectively, on their respective FPGA board, as illustrated in Table I. On the other hand, throughput density (η_{PE}) is an important figure-of-merit that determines the hardware efficiency of inference engine and it is given by number of operations performed by a single PE per unit time (i.e. OPs/PE). Improvements in Ω, σ and f_{clk} values of the proposed KPU are responsible for enhancing the η_{PE} value which is 2.26×, 2.20×, 2.3875× and 3.68× higher than the η_{PE} values of [11], [12], [20], and [21] reported implementations, respectively, as presented in Table I.

As discussed in Section I, memory footprint of inference engine mainly determines its overall power consumption. Hence, it is desirable to have an architecture with reduced memory footprint and efficient reuse of local data to alleviate the power requirement. Referring to Section III-A where it is described that the proposed KPU architecture simultaneously shares a single line memory with n numbers of PEs as well as it reuses each input data for $z(\alpha-s)^2\times$ and filters the values for $(A-s)^2\times$. As a result, the suggested KPU architecture significantly reduces the memory footprint in the form of block RAM (BRAM) usage in the FPGA. It uses a total of 64 BRAMs where each of them has a size of 36 kb and such memory utilization is 7.59×, 7.29×, 6× and 10× lesser than the utilizations reported by [11], [12], [21], and [22], respectively, as presented in Table I. It also shows that the energy efficiency (i.e. number of computations performed per watt of total power consumed) of our KPU is 21.79×, 15.09×, 5.09×, and 3.40× higher compared to the reported works from [11], [12], [21], and [22], respectively. Furthermore, suggested KPU consumes significantly lesser LUTs and flip-flops compared to the reported works, with an exception of flip-flop usage of [11] where our KPU utilizes slightly more flip-flops in comparison to [11]. However, the proposed KPU has a notably lesser utilization of area expensive BRAMs and LUTs compared to [11], as presented in Table I.

B. Proposed Hardware-Efficient Architecture of Classify Unit (CU)

As discussed earlier in Section II, to determine the class no. of detected object in the input image, computation of classification layer from the proposed Algorithm 1 involves searching and finding the largest activation value (i.e. AC_{Max}) from the FC_{last} layer, and the class no. (CN) associated with AC_{Max}. Hence, the hardware design for such activation search operation can be carried out by adopting either parallel or resource-shared approach. For a CNN model with N classes, the former parallel approach can be beneficial, provided all N different activation values from the FC_{last} layer of the CNN model are simultaneously available. Such approach requires an array of $N-1$ comparators as well as $2\times(N-1)$ number of 2:1 multiplexers in order to perform the search operation, and a highly-complex class no. generator (CNG) to produce N parallel CNs. However, for a CNN model with M neurons in the FC_{last} layer, it would require M number of PEs in the KPU to compute one AC item per clock cycle. In addition, a model with N classes typically comprises of N neurons in the FC_{last} layer. Thus, it requires $N\times M$ number of PEs in the PE-array of KPU for computing all N number of AC values. The proposed KPU architecture in this work has been designed using 864 PEs. Hence, our KPU is incapable of generating more than one AC value per clock cycle for a 1000 class image-classification models like VGG, GoogleNet and MobileNet, while computing the FC_{last} layer. Such adverse incapability is also present in the contemporary implementations of KPU architecture, reported in the literature [11], [20], [21], [22], and [23]. Therefore, rather adopting the parallel approach, this work focusses on the resource-shared approach for designing CU architecture, as shown in Fig. 7. It represents the proposed hardware-efficient micro-architecture of CU for the suggested simplified classification-technique, referring Algorithm 1. Major sub-modules of this architecture are data-&-signal router (DSR), classify unit controller (CUC), class no. generator (CNG) and activation searching unit (ACSU). Instead of using the approximated hardware for complex exponential and divisive operations [13], [14], [17], [18], the proposed CU architecture

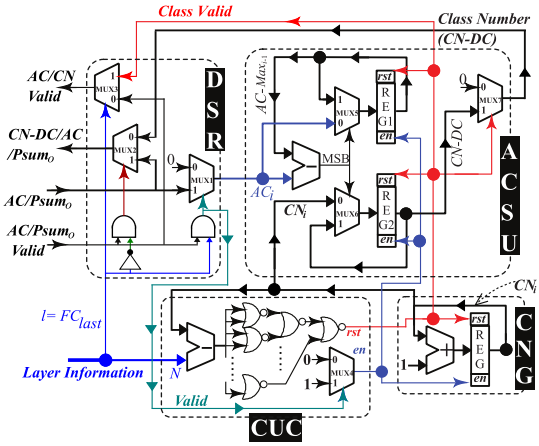

Fig. 7. Proposed micro-architecture of classify unit for our CNN inference engine.

has been designed using only comparators and steering logics (i.e. multiplexers). Suggested micro-architectures of the aforementioned submodules are comprehensively presented in the remaining portion of this subsection.

1) Micro-Architecture for DSR: Based on the discussion from Section II, KPU output has been directly fed to CU, irrespective of the layer being processed. In the proposed CU architecture, the key role of DSR is to collect both output (AC/P_{sum_o}) and validation (*valid*) signals from PE-array and KPC of KPU, respectively. Here, if the current layer is FC_{last} layer of CNN model (i.e. $l = FC_{last}$) then DSR routes them as AC_i activation-value to ACSU through multiplexer MUX1 and input-valid port of the CUC (referring line no. 25 of Algorithm-1), as shown in Fig. 7. It also shows that DSR collects the class no. of detected class (i.e. CN-DC), as the final classification result from ACSU, and the output validation signal from CUC that are routed through MUX2 and MUX3, respectively, to IEC of the inference engine. On the other hand, if $l \neq FC_{last}$ then it directly returns the incoming output and validation signals from KPU to IEC via MUX2 and MUX3, respectively, referring line no. 28 of Algorithm-1.

2) CUC and CNG Micro-Architectures: The suggested CU architecture operates only when the KPU is processing FC_{last} layer of CNN model; otherwise, DSR directly returns activation or P_{sum_o} to IEC, and all other modules of CU remain idle. Here, CUC manages the operations of two modules: CNG and ACSU by continuously monitoring the layer information, validity of activation from KPU, and the status of CNG. Such CUC architecture has been designed with a subtractor-based comparator and a 2:1 multiplexer, as shown in Fig. 7. When DSR indicates (using the valid signal) that KPU is processing FC_{last} layer by producing a valid activation AC_i value to CU and subsequently, the output of MUX4 in CUC activates the ACSU to process such AC_i value. In parallel, CUC also activates CNG to generate a *CN* value for the same input AC_i value. Further, comparator of CUC continuously compares the count of *CN* passed from CNG to ACSU with the total number of classes (i.e. N) in the CNN model that is derived from the layer information. Once CUC detects that all N classes have been covered, it deactivates both ACSU and CNG, as well as resets these modules to prepare them for subsequent inference task.

While processing the FC_{last} layer, KPU is configured in a way that it computes AC_i values in the same order of classes as in the ImageNet dataset that is used in this work. Thus, as presented in Fig. 7, CNG has been designed using a synchronous up-counter, wherein it linearly generates a CN_i for each of the AC_i values from FC_{last} layer.

3) ACSU: Referring the discussion from Section II, the computation of classification layer in Algorithm 1 involves searching and finding the AC_{Max} value, from the FC_{last} layer, and *CN* value associated with AC_{Max}. Here, the suggested ACSU architecture has been designed to generate *CN* value of the detected object in input image. Our ACSU micro-architecture comprises of a comparator (realized using subtractor), three 2:1 multiplexers (MUX5, MUX6 and MUX7), and two feedback registers: REG1 and REG2, as shown in Fig. 7. To begin with, both REG1 and REG2 are reset to null value that initializes both AC_{Max} and *CN-DC* values to zero, referring line no. 1 of Algorithm 1. While KPU architecture is processing the FC_{last} layer, if ACSU receives a valid AC_i then it is fed to both MUX5 and comparator. At the same time, CNG produces a class no. (CN_i) associated with the AC_i and is fed to MUX6, as presented in Fig. 7. In addition, the values of $AC_{Max_{i-1}}$, and $CN\text{-}DC_{i-1}$ from REG1 and REG2 are also fed to other terminals of MUX5 and MUX6, respectively. If the comparator finds current AC_i value is higher than the earlier largest value of activation $AC_{Max_{i-1}}$, then comparator output drives MUX5 and MUX6 to store AC_i and CN_i in REG1 and REG2 as the largest found-value of $AC_{Max_{i-1}}$ and the class no. associated with largest activation (*CN-DC*), respectively. On the other hand, if the comparator determines new activation value to be smaller than previously found largest activation value (i.e. $AC_i < AC_{Max_{i-1}}$), then it retains the previous value of REG1 and REG2, respectively, referring line nos. 25-26 of Algorithm 1. This process continues until CUC stops ACSU at the end of comparing all N activation values that are being fed from KPU. Eventually, it returns the *CN* associated with the largest activation to the DSR, which produces it as the classification result (i.e. class no. of the detected object *CN-DC*) and routes to IEC of our inference engine. In the aforementioned way, the proposed CU architecture classifies the object from the activation of FC_{last} layer without requiring any complex exponential and divisive computations.

C. Hardware Resources and Latency Analysis

FPGA implementation results of the proposed CU architecture and their comparison with the state-of-the-art works are presented in Table II. As discussed earlier in Section II-B, the complexity of classification layer surges with the number of classes (i.e. N) in the model. Thus, in the conventional state-of-the-art designs of CU for such classification layer [13], [16], [17], [18], both hardware-complexity and latency gradually increases with the N value. Hence, most of these reported works have fixed the magnitude of N between 8 to 10. Nevertheless, only [14] has reported the implementation results for N values of 10 and 1000, as listed in Table II. Therefore, comparison plots of the proposed CU architecture with respect to this state-of-the-art work from [14] has been presented in

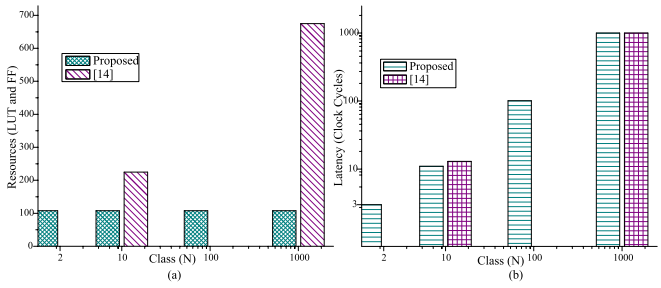

Fig. 8. Comparison analysis plots of (a) hardware resources and (b) latency, consumed by proposed and state-of-the-art CU architectures for different values of N.

TABLE II
COMPARISON OF PROPOSED CLASSIFY UNIT WITH RELEVANT STATE-OF-THE-ART WORKS

Work	FPGA Board	Prec. (bit)	N	Clock Freq. (MHz)	Θ_T (MIPs)	Resource LUTs, FFs	LUTs +FFs	α
[14]	ZC706	16	10 (1000)	500	500	128, 97 (561, 111)✠	225 (672)	2.22 (0.74)✠
[18]	KC705	16	10	154	154	2229, 224	2453	0.06
[13]	ZC706	16	8	500	500	395, 498	893	0.56
[17]	Zynq-7	16	10	-	-	1746, 1386	3180	-
Prop.	**ZCU102 ZC706 KC705**	**16**	**2–1000**	**1250 389 381**	**1250 389 381**	**59, 49**	**108**	**11.57 3.6 3.53**

α=Θ_T/{LUTs + FFs} MIPS/Resource where Θ_T: Computation Throughput.
✠: Quantifications of Hardware Resources and Throughput Density (α) for N=1000.

Fig. 8. It shows the comparison analysis in terms of latency and hardware consumption for different values of N.

Since the proposed CU architecture from Fig. 7 is independent of N, its hardware resources remain unchanged until N exceeds 2^k where k denotes the bit precision. Unlike, for state-of-the-art work [14], hardware consumption increases from 225 units for N=10 to 672 units for N=1000 (i.e. 2.99× surge), as shown in Fig. 8 (a). As a result, throughput density (i.e. α) falls from 2.22 to 0.74. Moreover, α value of our CU architecture is 38.33% and 79.44% higher than α values of [14] for N=10 and N=1000, respectively, as presented in Table II. However, classification latency of the proposed architecture increases linearly with N, as shown in Fig. 8 (b). Since the ACSU of our CU architecture contains single comparator, this resource-shared architecture demands a classification latency of N+1 clock cycles and it is still comparable to the latency of N+3 clock cycles, which is delivered by [14], as shown in Fig. 8 (b). Even though, there is such linear dependency of latency with N, the proposed architecture classifies images at the frame rate of 1.25×10^6 fps when N=1000 that is $41.67 \times 10^3\times$ faster than the conventional video frame-rate of 30 fps.

IV. HARDWARE DEVELOPMENT AND VALIDATION OF COMPLETE INFERENCE ENGINE

Referring to Fig. 1, the proposed efficient hardware-architectures of KPU and CU from Section III are aggregated into a complete CNN inference-engine which is implemented on FPGA board, and ASIC synthesized as well as post-layout simulated in 28 nm FD-SOI technology node. Here, fixed-point bit-quantization of $Q_{n.m}$=16 bit has been used for both ASIC and FPGA implementations. Table III

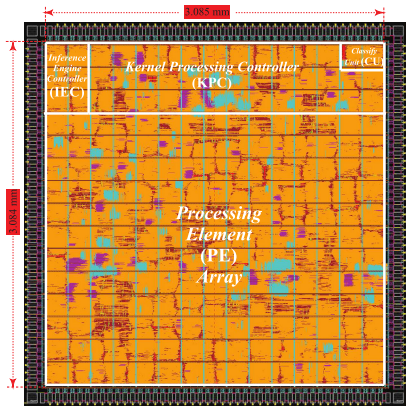

Fig. 9. ASIC chip-layout of the proposed inference engine for objection detection application in 28 nm FD-SOI technology node.

TABLE III
IMPLEMENTATION RESULTS OF THE COMPLETE INFERENCE ENGINE IN FPGA AND ASIC PLATFORMS

ZCU102 FPGA (16 nm FinFET Technology)		**ASIC (28 nm FD-SOI Technology)**	
Clock Freq.	1.25 GHz	Clock Freq.	3.85 GHz
N_{PE}	864	N_{PE}	864
Θ_T	2.16 TOPs	Θ_T	6.65 TOPs
LUT Count	97589	Cell Count	9964573
FF Count	27790	Core Area	9.5146 mm^2
BRAM Count	64	Core Dimen.	3.084×3.085 mm
Total Power	4.531W	Total Power	19.748 W
η_{PE}	2.5 GOPs/PE	η_{PE}	7.7 GOPs/PE
Energy Effn.	476.72 GOPs/W	Energy Effn.	336.74 GOPs/W

presents the quantifications of various design parameters of the proposed CNN inference-engine in both FPGA and ASIC platforms. On observing Table III, three major modules of our inference engine: CU, IEC and KPU consume 0.086%, 0.693% and 99.219% of the total hardware requirement, respectively. In addition, ASIC die-layout of the proposed inference engine in 28 nm FD-SOI technology node is shown in Fig. 9. Furthermore, this ASIC chip-layout is capable of delivering the throughput of 6.65 TOPs, as illustrated in Table III. On the other side, performing computation with the aid of suggested CU architecture (in the FPGA platform) alleviates operation time for classification layer from 2.05 ms in the ARM Cortex A-53 CPU to 0.8μs in the CU for a N=1000 class CNN-model, at the expense of 0.086% extra hardware. To the best of authors' knowledge, such complete implementation of CNN inference engine that specially includes the hardware version of CU, along with KPU which performs all type of computation for the feature extraction layers of state-of-the-art CNN models, has been first time reported in our paper. Furthermore, such inference engine FPGA-prototype is used for the object classification and validated for its correct functionality.

A. Real-World Test Setup for Functional Validation

As discussed in Section II-A, an object classification system comprises of proposed CNN inference engine, on/off-chip processor, and DRAM. For implementing such system, this work adopted the Xilinx ZCU102 MPSoC FPGA-board that has a processing system (containing an ARM-cortex processor with six cores), a 4.5 GB DDR4 DRAM, and a user-programmable

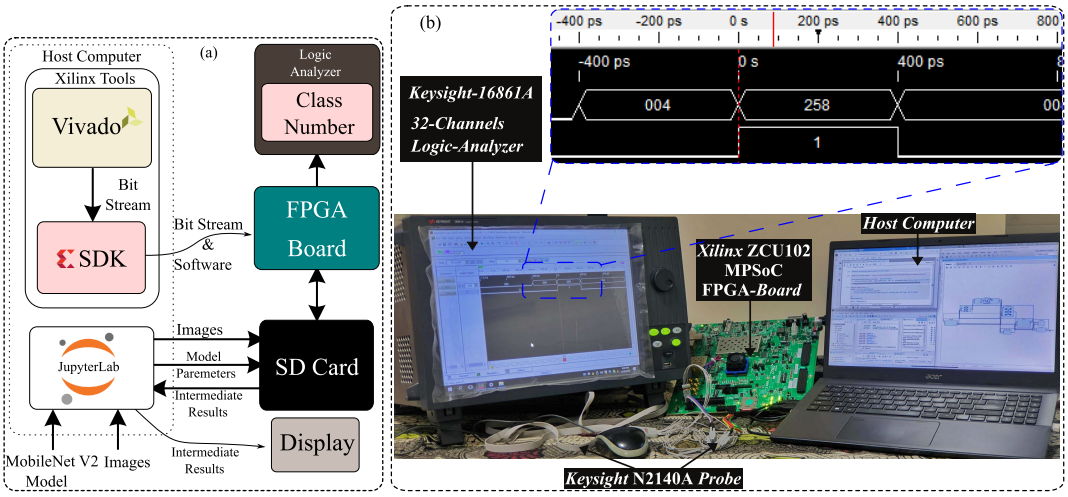

Fig. 10. Real-world test setup of object detection system for functional validation of the proposed CNN inference engine.

FPGA fabric inside the same system-on-chip. Here, Fig. 10 (a) and (b) show a schematic representation of the test setup and a snapshot of the actual prototype, respectively. Our test setup contains host computer, SD memory card, FPGA-board, and Keysight-16861A 32-channels logic-analyzer, along with connecting probe (Keysight N2140A probe), as shown in Fig. 10 (b). Here, the host computer is used for developing the HDL code for suggested hardware architectures, integrating them, synthesizing and generating the bit-stream. It is also used for creating necessary software to program and operate the FPGA board. Furthermore, host computer imports the CNN model and provides extracted model parameters as well as input image to the FPGA board. For the aforementioned purposes, host computer is installed with two Xilinx tools: Vivado 2018.2 and SDK 2018.2, and a Python tool (JupyterLab 3.0.14), as shown schematically in Fig. 10 (a). Using such Vivado tool, an AXI4-stream compatible intellectual-property (IP) has been developed from the Verilog HDL code of our inference engine that is interfaced with an IP of a memory-mapped AXI direct-memory-access (DMA) module. This DMA module is interfaced with one of the accelerated cache-coherent AXI-HP ports of the processing system to form a high band-width link between on-chip processor and inference engine for high speed access of on-board DDR4 DRAM. Configuration bits and status signals of the inference engine are interfaced using the general purpose AXI-lite ports of the processing system with the aid of memory-mapped AXI4 registers. Output of the inference engine, implemented on FPGA board, are also tapped via its peripheral module interface (PMOD) in order to show the classification result on the screen of logic analyzer.

Furthermore, an interrupt signal has been interfaced between processing system and inference engine to keep the processor free, rather than halting it during the computation in the inference engine. Thereafter, referring Fig. 10 (a), synthesized bit stream of inference engine is exported to Xilinx SDK 2018.2 tool in order to develop necessary software that programs the FPGA board. Simultaneously, a CNN model has been imported into the Python environment of JupyterLab 3.0.14 tool that optimizes the model and extracts the model parameters (like weights and biases), and stores them as binary files (with '.bin' format) in the SD memory, as presented in Fig. 10 (a). Additionally, using the aforementioned Python environment, pixel information of input images are extracted and stored in the same SD memory card, as binary files, as illustrated in Fig. 10 (a). Following that, such SD memory card is removed from the host computer and installed in the FPGA board. To read these model parameters and input images from the SD memory card in the FPGA board, we have developed a stand-alone software for the FPGA board in the Xilinx SDK tool. When executed in the FPGA board, after reading the data from SD memory card, this stand-alone software loads them in the DDR4 DRAM of FPGA board and also sets the configuration bits for inference engine through the memory-mapped AXI4 registers, connected to the AXI4-lite ports. In order to visualize the intermediate results while processing the computation of a CNN model, instructions have been included in this software to store some of the activations from intermediate layers of model into the SD memory card. One such example has been shown in Fig. 11 where intermediate results from a CNN model has been visualized.

Once the software is ready, FPGA board is connected to the host computer. Following that, PMODs of FPGA board are connected to the channels of Keysight-16861A 32-channels logic-analyzer, via Keysight N2140A probe, as shown in Fig. 10 (b). With the aid of Xilinx SDK tool in the host computer, FPGA board has been programmed and the compiled file (with '.elf' format) of software has been executed on one of the hexa cores of ARM processor in the processing system of ZCU102 MPSoC FPGA-board. Hence, the final classification results are displayed on the logic analyzer in the form of class number of the detected class (i.e. *CN-DC*). Class labels of the class numbers that are detected for three of the tested images are presented in the rightmost 3×3 matrix in Fig. 11, and the class labels for all one-thousand different class numbers for 1000-class ImageNet models are provided here [24]. Furthermore, activations of the intermediate layers are stored in the SD memory card that are later plotted using

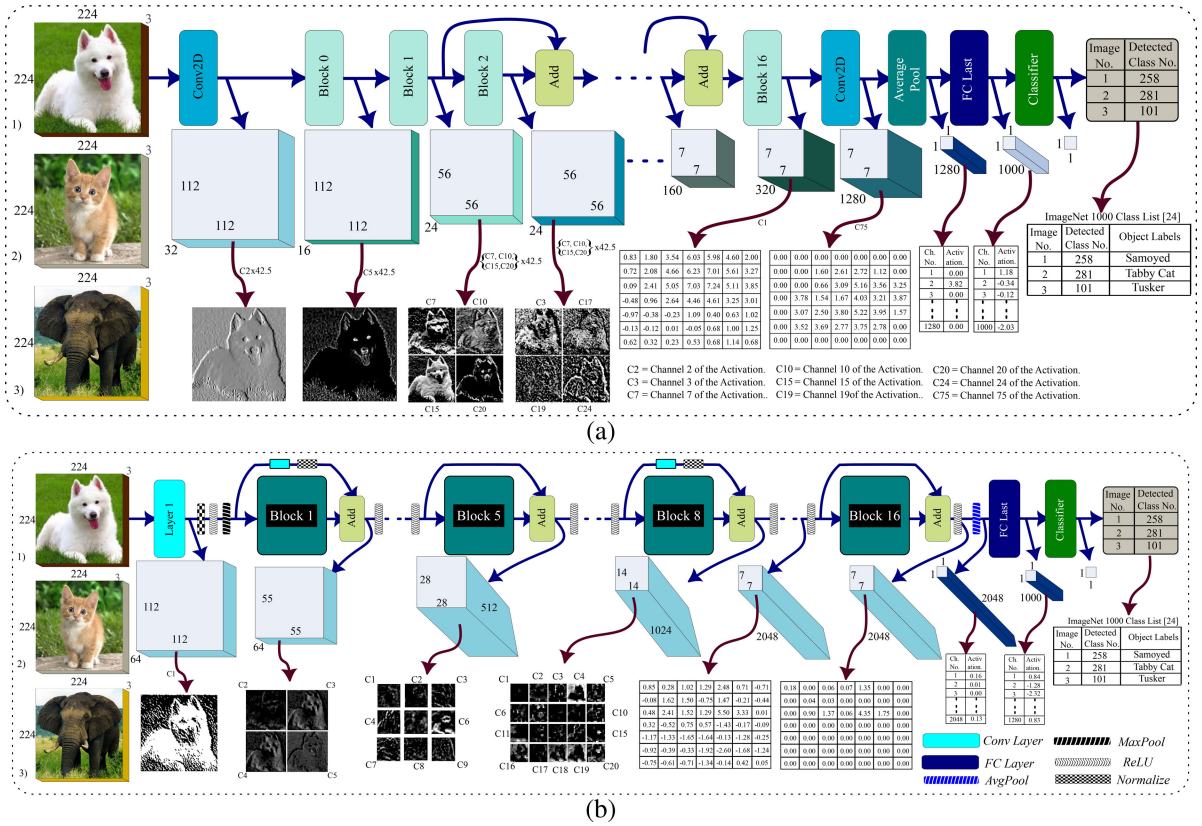

Fig. 11. Schematic representations of layer-wise processing of images in the proposed CNN inference engine using (a) MobileNet-V2, and (b) ResNet50 models.

Python simulation environment in the host computer, as shown in Fig. 11.

B. Model Compatibility

As discussed earlier in Section III-A.5 and III-A.1, the proposed architecture intrinsically supports 1×1, 3×3, 5×5, and 7×7 kernels, as well as all computations for *Conv*, *FC*, *MaxPool*, *AvgPool*, *ReLu*, and *ReLu6* layers. Therefore, our architecture can process any state-of-the-art CNN models that use any combination of these kernel sizes and computation layers. Models that have been used to validate our inference engine include VGG, ResNet, GoogLeNet, MobileNet, and EfficientNet. Following section discusses the detailed implementation flows for two of the extremely-complex CNN models: MobileNet-V2 and ResNet50.

C. Validation and Implementation Results

1) MobileNet-V2 Implementation Flow*:* In MobileNet-V2 model architecture, except first convolution, last convolution, average pool and fully-connected layers, the remaining layers use a block-based approach with or without residual connections. Here, each block consists of a 1×1 expansion layer which is followed by ReLU6, a 3×3 depth-wise separable layer that is also followed by ReLU6, and a 1×1 projection layer. Here, 1×1 expansion layer expands the number of channels to multiple fold, before feeding the activation to 3×3 depth-wise separable layer that produces activation with same number of channels and finally, projection layer compresses the number of channels. Thus, if the CNN inference engine processes one complete layer at a given time then it requires massive data movement between off-chip DRAM and on-chip inference engine. Such data movement is comparatively lesser on processing the entire block together, as the number of channels for input and output of a block is significantly smaller. Since, PE-array size and memory inside the proposed CNN inference engine is limited to fit all computations and data of a block of MobileNet V2 model, we have adopted the pipelined approach where the PE array is segregated into three groups. Now, a group of PEs has been configured to perform the computation of 1×1 expansion layer and returns the activations to IEC of our inference engine (refer Fig. 1) that stores them to the line memories of second group in the PE-array of KPU. Simultaneously, such second group of PEs performs the computation for 3×3 depth-wise separable layer, while the third group of PEs carries out computation for 1×1 projection layer. Finally, IEC stores the activations of that block, back to DRAM when all the line memories are occupied. As a result, it significantly reduces the number of off-chip data movements. The number of PEs allotted to each of these groups are decided according to the ratio of their computational load. However, such ratio between these groups varies from block to block. Thus, we have also included additional instructions in software to set the configuration bits for dynamically allotting different number of PEs to these groups, according to the variations of computation ratios in different blocks.

To validate the functionality of our CNN inference-engine, several images have been fed into the FPGA prototype of an object classification system that uses our CNN inference engine. Three of such input images and their classifica-

tion results are presented in Fig. 11 (a). As discussed in Section IV-A, the final classification result has been displayed on the keysight logic analyzer in the form of *CN-DC*, while activations of the intermediate layers are stored in the SD memory card. However, for better clarity, we have presented detailed view of these layer-wise results for one image in Fig. 11 (a), and the snapshot of its final classification result on the screen of logic analyzer in Fig. 10. Nevertheless, final classification result for all three images are included in Fig. 11 (a). As discussed in Section IV-C.1, MobileNet V2 model uses blocks to combine 1×1 expansion layer, 3×3 depth-wise separable layer, and 1×1 projection layer. Therefore, Fig. 11 (a) displays the output of only projection layer, as the final output of a block, rather than displaying for each layer inside the block. Here, dimension of each channel of the output activation for Layer-0, Block-0, Block-1, Block-2, and Block-3 are 112×112, 112×112, 56×56, 56×56 and 56×56, respectively. Thus, these activations are too large to visualize as a matrix in Fig. 11 (a). Therefore, we plotted them as a single-channel gray-scale images. However, each convolution and expansion layer use ReLU6 at the output to clip the output values below the numerical value of 6. Thereby, we used a scaling factor of 42.5 in the python environment to scale up the values before plotting them as grey-scale pixels. Nevertheless, each output channel of block 13 to block 16 and the last convolution layer is a 7×7 matrix. Since the dimensions of these channels are small enough to visualize as matrix, Fig. 11 (a) displays their values in the form of a 7×7 matrix. Here, each value in the matrix formed by a channel of the output activation denotes the extracted score for the presence of a feature on the input image that has been searched by a fitter associated with that feature. The activations of average-pool and fully-connected layers are one dimensional array whose preview is presented in Fig. 11 (a). Therefore, output of the classification layer is a single value which is the class no. of the object that has been detected by the proposed CNN inference-engine from the input image. Hence, the human readable labels that are associated with these class nos. are provided with the ImageNet class-1000 dataset [24].

2) ResNet50 Implementation Flow*:* Except the first convolution layer and the last FC-layer, remaining computations of the ResNet50 model is distributed over 16 different computational blocks, with each block having a residual connection between its input and output. While majority of the residual connections simply add the input of a block with its output. Some residual connections also have a convolution layer embedded in it (like block 1, 4, 8 and 14). Each of these 16 computational blocks is made of a 1×1 convolution layer, followed by a 3×3 convolution layer, which is followed by another 1×1 convolution layer. Computation of each of the convolution layers (both inside a block, outside a block, and in residual connections) is followed by a normalization and a *ReLU* operation. While computation of *Conv*, *FC*, *ReLU* and *MaxPool* operations are handled by dedicated hardware blocks in the PE array, offset subtraction for the normalization operation here has been achieved by supplying 2's complement form of the offset values through B, P_{sum_i} port of the PE array in Fig. 3. Here, scaling operation has been performed using the bit shifting approach, similar to the way of average pooling, as discussed in Section II-B.

When processing a block, significant portion of the output feature map (up to 2.25 Mb) is reused within PE array for the subsequent layer. Towards the end of computation of a block, it adds the output feature map with the output of residual connection. Since the residual connections require whole output feature map of the previous block/layer, and the output of residual connection is used only after the computation of eight different computations in a block. Therefore, we copy the final output feature-map of a block to outside the PE array and hold them until we need to send them back to the PE array for residual additions. Therefore, in Fig. 11 (b), output of a complete block after the residual addition has been presented rather than showing the output of each individual layers inside these blocks.

In this ResNet50 model, dimension of each channel of the output feature map is 112×112 for layer 1. It is 55×55 for blocks 1−3, 28×28 for blocks 4−7, 14×14 for blocks 8−13, and 7×7 for blocks 14−16. Thus, some channels of the output feature map for layer 1 to block 16, as a grayscale image, has been shown in Fig. 11 (b). Average pool after block 16 reduces the 7×7 dimension to 1×1. As a result, both input and output feature maps of the FC layer is an one dimensional (1D) array. A preview of the same is presented in Fig. 11 (b). Finally, output of the classification layer is a single value, which is the class number of the object that has been detected by the proposed CNN inference engine from the input image. Human readable labels that are associated with these class numbers are provided with the ImageNet class-1000 dataset [24]. Here, for both models, it is shown that the detected class numbers of three images along with their object labels. These class numbers produced by the proposed inference engine correctly depicts the object present in the input image.

V. Conclusion

This work mainly contributed towards the designs of efficient algorithm and hardware-architectures for the complete CNN inference-engine. Significant factors like energy-efficiency and throughput of such engine were enhanced that made it suitable for edge applications. Accuracy analysis of the proposed algorithm was comprehensively carried out. Detail implementations of various proposed architectures as well as the complete CNN inference engine were performed in both FPGA and ASIC platforms. Furthermore, FPGA prototype of our CNN inference engine was functionally verified in real-world test environment using the well-known CNN model. Eventually, our implementation results showed promising outcomes in comparison to the contemporary implementations and hence, these suggested designs will play crucial role in various edge applications.

References

[1] Z. Xiao, P. Xu, X. Wang, L. Chen, and F. An, "A multi-class objects detection coprocessor with dual feature space and weighted softmax," *IEEE Trans. Circuits Syst. II, Exp. Briefs*, vol. 67, no. 9, pp. 1629–1633, Sep. 2020.

[2] M. Alam, M. D. Samad, L. Vidyaratne, A. Glandon, and K. M. Iftekharuddin, "Survey on deep neural networks in speech and vision systems," *Neurocomputing*, vol. 417, pp. 302–321, Dec. 2020.

[3] Y. Alghofaili, A. Albattah, and M. A. Rassam, "A financial fraud detection model based on LSTM deep learning technique," *J. Appl. Secur. Res.*, vol. 15, no. 4, pp. 498–516, Oct. 2020.

[4] R. Ravindran, M. J. Santora, and M. M. Jamali, "Multi-object detection and tracking, based on DNN, for autonomous vehicles: A review," *IEEE Sensors J.*, vol. 21, no. 5, pp. 5668–5677, Mar. 2021.

[5] Md. N. Islam, R. Shrestha, and S. R. Chowdhury, "A new hardware-efficient VLSI-architecture of GoogLeNet CNN-model based hardware accelerator for edge computing applications," in *Proc. IEEE Comput. Soc. Annu. Symp. VLSI (ISVLSI)*, Jul. 2022, pp. 414–417.

[6] Y.-H. Chen, T. Krishina, J.-S. Emer, and V. Sze, "Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks," *IEEE J. Solid-State Circuits*, vol. 52, no. 1, pp. 127–138, Nov. 2016.

[7] V. Sze, Y.-H. Chen, T.-J. Yang, and J. S. Emer, "Efficient processing of deep neural networks," *Synth. Lect. Comput. Archit.*, vol. 15, no. 2, pp. 1–341, 2020.

[8] C. Wu, J. Zhuang, K. Wang, and L. He, "MP-OPU: A mixed precision FPGA-based overlay processor for convolutional neural networks," in *Proc. 31st Int. Conf. Field-Program. Log. Appl. (FPL)*, Aug. 2021, pp. 33–37.

[9] F. Spagnolo, S. Perri, F. Frustaci, and P. Corsonello, "Reconfigurable convolution architecture for heterogeneous systems-on-chip," in *Proc. 9th Medit. Conf. Embedded Comput. (MECO)*, Jun. 2020, pp. 1–5.

[10] D. Wang, K. Xu, J. Guo, and S. Ghiasi, "DSP-efficient hardware acceleration of convolutional neural network inference on FPGAs," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 39, no. 12, pp. 4867–4880, Dec. 2020.

[11] M. N. Islam, R. Shrestha, and S. R. Chowdhury, "An uninterrupted processing technique-based high-throughput and energy-efficient hardware accelerator for convolutional neural networks," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 30, no. 12, pp. 1891–1901, Dec. 2022.

[12] L. Xuan, K.-F. Un, C.-S. Lam, and R. P. Martins, "An FPGA-based energy-efficient reconfigurable depthwise separable convolution accelerator for image recognition," *IEEE Trans. Circuits Syst. II, Exp. Briefs*, vol. 69, no. 10, pp. 4003–4007, Oct. 2022.

[13] D. Zhu, S. Lu, M. Wang, J. Lin, and Z. Wang, "Efficient precision-adjustable architecture for softmax function in deep learning," *IEEE Trans. Circuits Syst. II, Exp. Briefs*, vol. 67, no. 12, pp. 3382–3386, Dec. 2020.

[14] F. Spagnolo, S. Perri, and P. Corsonello, "Aggressive approximation of the SoftMax function for power-efficient hardware implementations," *IEEE Trans. Circuits Syst. II, Exp. Briefs*, vol. 69, no. 3, pp. 1652–1656, Mar. 2022.

[15] Y. Cao, W. Xiao, J. Jia, D. Wu, and W. Zhou, "Cordic-based softmax acceleration method of convolution neural network on FPGA," in *Proc. IEEE Int. Conf. Artif. Intell. Inf. Syst. (ICAIIS)*, Mar. 2020, pp. 66–70.

[16] Y. Zhang et al., "Base-2 softmax function: Suitability for training and efficient hardware implementation," *IEEE Trans. Circuits Syst. I, Reg. Papers*, vol. 69, no. 9, pp. 3605–3618, Sep. 2022.

[17] K. Chen, Y. Gao, H. Waris, W. Liu, and F. Lombardi, "Approximate softmax functions for energy-efficient deep neural networks," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 31, no. 1, pp. 4–16, Jan. 2023.

[18] Y. Gao, W. Liu, and F. Lombardi, "Design and implementation of an approximate softmax layer for deep neural networks," in *Proc. IEEE Int. Symp. Circuits Syst. (ISCAS)*, Oct. 2020, pp. 1–5.

[19] M. N. Islam, R. Shrestha, and S. R. Chowdhury, "Low-complexity lassification technique and hardware-efficient classify-unit architecture for CNN accelerator," in *Proc. 37th Int. Conf. VLSI Design 23rd Int. Conf. Embedded Syst. (VLSID)*, Jan. 2024, pp. 210–215.

[20] A. A. Gilan, M. Emad, and B. Alizadeh, "FPGA-based implementation of a real-time object recognition system using convolutional neural network," *IEEE Trans. Circuits Syst. II, Exp. Briefs*, vol. 67, no. 4, pp. 755–759, Apr. 2020.

[21] J. Li, K.-F. Un, W.-H. Yu, P.-I. Mak, and R. P. Martins, "An FPGA-based energy-efficient reconfigurable convolutional neural network accelerator for object recognition applications," *IEEE Trans. Circuits Syst. II, Exp. Briefs*, vol. 68, no. 9, pp. 3143–3147, Sep. 2021.

[22] J. Qiu et al., "Going deeper with embedded FPGA platform for convolutional neural network," in *Proc. ACM/SIGDA Int. Symp. Field-Programmable Gate Arrays*, Feb. 2016, pp. 25–35.

[23] Y.-X. Chen and S.-J. Ruan, "A throughput-optimized channel-oriented processing element array for convolutional neural networks," *IEEE Trans. Circuits Syst. II, Exp. Briefs*, vol. 68, no. 2, pp. 752–756, Feb. 2021.

[24] *Kagle Dateset: ImageNet-1000-Classes*. Accessed: Sep. 23, 2023. [Online]. Available: https://www.kaggle.com/datasets/skyap79/image net-classes

Md. Najrul Islam (Graduate Student Member, IEEE) received the Bachelor of Engineering degree in electronics and telecommunication engineering from Tripura University, Agartala, India, in 2016, and the M.Tech. degree in VLSI from the National Institute of Technology Meghalaya, India, in 2018. He is currently pursuing the Ph.D. degree with the School of Computing and Electrical Engineering, Indian Institute of Technology Mandi, India. His current research interests include developing VLSI architectures of energy and area-efficient reconfigurable deep-neural-network hardware accelerators for edge applications.

Rahul Shrestha (Senior Member, IEEE) received the Bachelor of Engineering degree in telecommunication engineering from the B.M.S. College of Engineering, Bengaluru, India, in 2008, and the Ph.D. degree in electronics and electrical engineering from Indian Institute of Technology Guwahati in 2014. Currently, he is an Associate Professor with the School of Computing and Electrical Engineering, Indian Institute of Technology Mandi. His research interests include digital VLSI architectures and its transformation into ASIC-chip or FPGA-prototype for the real-world applications of signal processing, wireless communication, deep neural networks, forward-error-correction channel decoders, cognitive radio, and cooperative spectrum sensing.

Shubhajit Roy Chowdhury (Senior Member, IEEE) received the Ph.D. degree from the Department of Electronics and Telecommunication Engineering, Jadavpur University, in 2010. He is currently an Associate Professor with the School of Computing and Electrical Engineering, Indian Institute of Technology Mandi. Previously, he was an Assistant Professor with the Centre for VLSI and Embedded Systems Technology, IIIT Hyderabad. He was a recipient of the University Gold Medals for his B.E. and M.E. degrees in 2004 and 2006, respectively; the Altera Embedded Processor Designer Award in 2007; and the winner of five best paper awards. He received the Award of the Fellow of Society of Applied Biotechnology (FSAB) by the Society of Applied Biotechnology in 2012. He received the Young Engineers' Award 2012–2013 by the Institution of Engineers, India, for his outstanding contribution in the field of electronics and telecommunication engineering.