HW/SW Co-Design of Cost-Efficient CNN Inference for Cognitive IoT

Kwangho Lee*, Joonho Kong*, Arslan Munir†
*School of Electronics Engineering, Kyungpook National University, South Korea
†Department of Computer Science, Kansas State University, Manhattan, KS
{lkh, joonho.kong}@knu.ac.kr, amunir@ksu.edu

***Abstract*—Cognitive Internet of things (IoT) is a novel paradigm that outfits the contemporary IoT with a "brain" to impart high-level intelligence. Convolutional neural networks (CNNs) are an integral part of cognitive IoT that support inference and decision-making. In this paper, we demonstrate a resource-efficient hardware/software (HW/SW) co-design of a CNN architecture for cognitive IoT. We only offload image-to-column (im2col) and general matrix multiply (GEMM), which are the most time- and energy-consuming part of convolution layer operations, to the field-programmable gate array (FPGA)-based accelerator. We also exploit the parallelism in the operations of convolution layers to efficiently hide a non-negligible portion of execution time required for bias and activation. Experimental results demonstrate the resource, performance, and energy efficiency of our HW/SW co-design. Results indicate a speedup of 1.3X$\sim$2.0X and energy reduction of 19.4%$\sim$44.3% as compared to using only a general-purpose processor.**

***Index Terms*—Convolutional neural networks, cognitive engine, Internet of things, hardware/software co-design, cost efficiency**

I. Introduction

Many of the emerging Internet of things (IoT) applications not only require the IoT devices to perform compute-intensive tasks but also to *learn*, *think*, and *understand* both physical and social worlds by themselves thus motivating the development of a new paradigm, named cognitive IoT [8]. Cognitive IoT outfits the current IoT with a "brain" for high-level intelligence. Cognitive IoT will be able to accomplish various application tasks including resource control (e.g., sensing resolution, actuation), inference, and decision-making with minimum human intervention/supervision thus saving human's time and effort and will also provide efficient resource usage. Hence, there is a need to develop a flexible, high-performance, energy-efficient, and cognitive IoT architecture to assist with various emerging IoT applications (e.g., agriculture, smart homes, smart cities, military, etc.).

Fig. 1 depicts our proposed cognitive IoT architecture. This architecture consists of a host processor which comprises of an application processor, co-processors, and accelerators. The host processor is connected to various low-power interface processors that collect data from sensors and control actuation elements. The core component of a cognitive IoT architecture is *cognitive engine*, which provides cognition capabilities to the IoT device. The cognitive engine implements various deep learning accelerators, such as convolutional neural networks (CNNs), recurrent neural networks (RNNs), and multilayer perceptrons (MLPs). The cognitive engine helps with various cognitive tasks in IoT devices, such as in-situ analysis of the sensed data, local resource management, inference, decision-making, and response calculation for the IoT devices based on the analysis of the sensed data. Increasing cyber-physical and vision applications of IoT make CNNs an integral part of cognitive engine for imparting real-time object detections and/or classifications. To enable efficient on-device CNN inferences, hardware-based CNN inference engines are often used instead of general-purpose central processing units (CPUs) or graphics processing units (GPUs). This work focuses on HW/SW co-design of CNN inference engine for cognitive IoT.

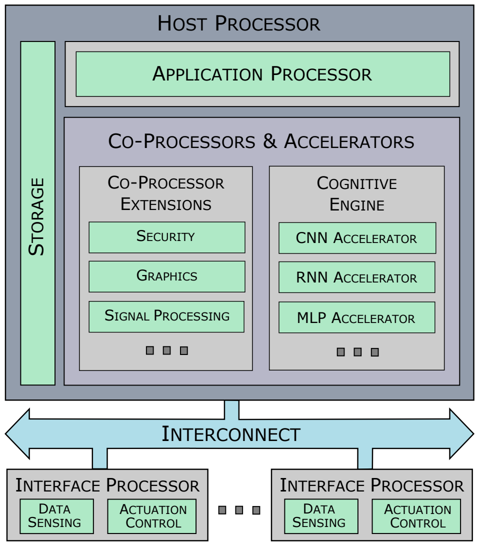

Fig. 1. Cognitive IoT architecture.

For cognitive IoT computing platforms, implementing the cognitive engine with application-specific integrated circuit (ASICs) would be desirable for better optimization; however, ASICs being hardwired provide no flexibility to adopt a new cognitive engine. Alternatively, field-programmable gate array system-on-chip (FPGA-SoC) that equips programmable logic, CPU, and other hard intellectual properties (IPs) provides an attractive platforms because of its reconfigurability, flexibility, cost efficiency, and faster time-to-market. *Cost efficiency* is an important metric to consider for real-world adoption of on-device CNN inference architectures in IoT devices. We define cost efficiency metric as: an architecture (or implementation) is cost-efficient if it requires fewer resources under performance and energy efficiency constraints as compared to other architectures (implementations). Though there have been many studies and proposals on fast and efficient CNN inference for FPGA platforms (e.g., [5], [6], [7], [9], and [10]), cost efficiency of the implementations has been largely ignored, which may make those designs infeasible for low-power and resource-constrained cognitive IoT.

In this paper, we demonstrate a new design approach for efficient on-device CNN inference in resource-constrained cognitive IoT devices. We prototype our proposed design for cost-efficient CNN inference on an FPGA-SoC. For cost-efficient design, our approach performs finer-grained offloading of the operations required for convolution layers instead of offloading the entire convolution layer operations. We choose im2col (image-to-column) and MAC (general matrix-multiply (GEMM) and accumulation), which we refer to as iMAC (im2col + MAC), for offloading to the hardware accelerator. Though only offloading GEMM could be a cost-efficient approach, we have determined that im2col significantly increases the amount of data transfer (e.g., $9\times$ in the case of 3×3 convolution), and thus necessitates in-situ execution of im2col and MAC within the accelerator. In addition to hardware acceleration of CNN inference, we further propose and implement two important software-level support mechanisms: (1) efficient channel partition and input/weight allocation, and (2) the pipelined execution of the front-end (e.g., input data transfer, im2col, and MAC) and the back-end part (e.g., bias addition and activation) of the execution. The efficient input channel partition, input/weight allocation, and the pipelined execution are essential parts to maximize the resource utilization in the system. Results demonstrate the superiority of our FPGA-based implementation for Tiny-Darknet [3] CNN model with full-stack software including operating systems (PetaLinux) and Darknet framework [2] in terms of response time and energy consumption as compared to the implementations using the CPU with NEON (SIMD (Single Instruction Multiple Data) architecture extension for the Arm Cortex processors) supports as well as the implementation that only offloads the GEMM to the hardware accelerator.

We summarize our contributions as follows:

- We propose a novel CNN accelerator for resource-constrained cognitive IoT leveraging HW/SW co-design that leads to much less hardware costs as compared to contemporary designs.
- We propose a HW/SW-cooperative optimization technique that exploits the parallelism in CNN convolution layer operations.
- We implement the prototype of our proposed design in ZED platform (which equips ZYNQ7020) with full-stack software including operating system (petaLinux).
- Experimental results exhibit that our design with eight iMAC processing elements (PEs) improves performance of the CNN inferences by $1.3\times\sim2.0\times$, while reducing energy consumption by 19.4%$\sim$44.3%, as compared to running CNN inference with a general-purpose processor.

The remainder of this paper is organized as follows. Section II presents relevant background and motivation for our work. Section III elaborates our proposed HW/SW co-design for convolution layer of CNN inference engine. The implementation of our HW/SW co-design is presented in Section IV. Experimental results are discussed in Section V. Finally, Section VII concludes this work.

II. Background and Motivation

Inference tasks of CNNs are often required to be executed on-device (e.g., IoT edge) because of limited communication bandwidth to cloud and security/privacy concerns. However, since IoT devices have a tight resource budget, it is very hard to meet the response time requirement of the CNNs. A major challenge of CNN inferences in resource-constrained IoT devices is to find an optimal point of trade-off between resource cost and response time. A tight budget for hardware resources makes it hard to offload the computational tasks such as convolution operations. In such systems, CNN inferences are typically performed by CPUs. However, CPUs are known to be inefficient for CNN execution because of their meager performance on data-parallel workloads such as matrix multiplications. Even with the single instruction multiple data (SIMD) instructions, general CPU provides only a few SIMD execution lanes that limit the exploitation of parallelism in matrix or vector operations. In addition, CPUs rely on caches for data transfer between the CPU and main memory, which can be inefficient when running CNNs with large number of weights (i.e., weight size $>$ cache capacity) because of cache misses. These cache misses can result in slower response time and relatively high energy consumption.

To improve response time and energy efficiency, one can employ the GEMM (GEneral Matrix Multiplication) hardware accelerator such as systolic arrays to offload convolution operations of CNNs. In fact, GEMM accelerators can be used not only for CNNs but also for many other embedded/IoT applications, making this design decision very attractive for resource-constrained IoT and embedded systems. As shown in Fig. 2, when running CNNs with a GEMM accelerator, the GEMM can be executed in a dedicated GEMM accelerator while the other CNN tasks can be executed in the CPU. However, before performing GEMM, we need to unroll the input feature maps (IFMs) to fit the data into matrix or vectors that can be applied to the GEMM hardware. This is often called as 'im2col' which is already employed in many CNN frameworks such as [2]. However, decoupled im2col and GEMM execution causes non-negligible data storage and transfer overhead. The main reason for this overhead is that im2col actually explodes the amount of storage requirements and the amount of data transfer depending on the weight kernel size (e.g., 3×3 or 5×5). For example, to execute 3×3 convolution operations with GEMM on the accelerator, we need to transform the IFM to multiple vectors that have redundant IFM elements resulting in $9\times$ more data storage and transfer requirements for IFMs. To utilize a GEMM accelerator, we need to offload the unrolled vectors, which has $9\times$ larger size as compared to the original data size, which is not desirable for meeting response time and energy constraints due to high data transfer overheads.

The other problem using the hardware accelerators for CNNs is that the CPU remains idle during the direct memory access (DMA) transfer and accelerator execution. If we could

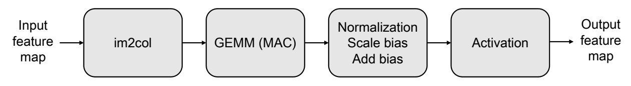

Fig. 2. A general flow of the convolution layer execution.

find the parallelism among the convolution layer operations, we would efficiently utilize the hardware resources in the resource-constrained systems. To exploit the parallelism, the approach that uses both accelerators and CPUs such as [4] would be beneficial. Our work also exploits the parallelism in convolution layer operations to utilize both the accelerator and the CPU simultaneously. The exploitation of parallelism and HW/SW co-design eventually leads to better response time and energy efficiency of CNN inference in our design.

III. Hardware/Software Co-Design in Resource-Constrained FPGA-SoC Systems

Our design is based on a hardware and software (HW/SW) co-design approach. Though there are several different layers in general CNN architecture (e.g., convolution layer, max pooling layer, etc.), we focus on accelerating the convolution layer under tight resource budgets.

A. Architecture Overview

In our design, we offload the im2col, GEMM, and accumulation (from here, we use the term 'MAC' and 'GEMM and accumulation', interchangeably) to the loosely coupled hardware accelerator (in programmable logic) which transfers the data via direct memory access (DMA). For internal memories for the hardware accelerator, we have block RAMs (BRAMs) that temporarily store the input, weight, and output for im2col and MAC. Input and weight BRAMs are to store the input feature map and weight data, respectively. The output BRAM stores the intermediate results of output feature map(s) to which the bias addition and activation has not yet been applied. As shown in Fig. 3, once the im2col and MAC operations are finished in the accelerator, the remaining back-end tasks (bias addition and activation) are performed by the CPU. Since the iMAC accelerator can be utilized simultaneously with the CPU (i.e., they can be used in parallel), we perform the im2col+MAC operations and back-end tasks in a pipelined manner. That means, during the execution of the im2col and MAC to generate the current output feature map, we can perform the bias addition and activation for the previous output feature map. This also eliminates a need of bias addition and activation logic in the hardware accelerator, resulting in resource-efficient implementation. For 1×1 convolution operations in a single input channel, we do not perform im2col but execute only MAC since we do not need to unroll input feature maps in the case of 1×1 convolution. Please note that we do not use Winograd convolution because of its high memory space requirements, which may not be suitable for resource-constrained systems.

B. iMAC Hardware Accelerator

Our hardware accelerator functions a combined operation of im2col and MAC (we refer to it as im2col+MAC or iMAC).

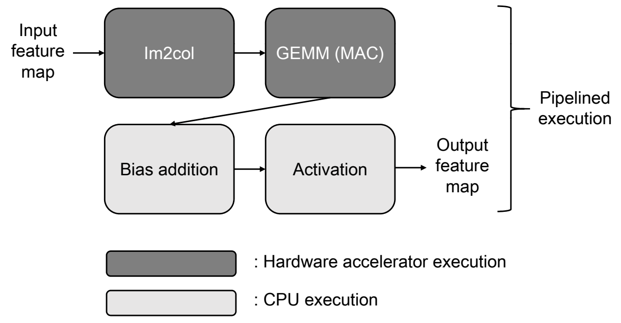

Fig. 3. Convolution layer execution flow of our design.

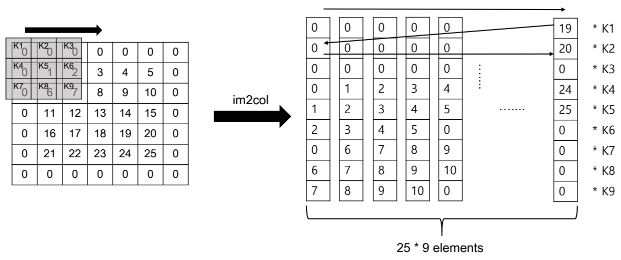

Fig. 4. Conventional im2col execution in software.

Fig. 4 shows the conventional software-based im2col with 5×5 input feature map, 3×3 weights, one zero-padding, and stride as one. Since the GEMM or MAC operations can be executed after the SW-based im2col is entirely finished, it just unrolls the input feature map into multiple vectors for matrix multiplication and accumulation in a row-wise fashion as shown in Fig. 4. However, as we discussed in Section II, the im2col actually generates 9× more data (25*9 elements) in the case of 3×3 convolutions, which causes inefficiency in terms of data storage and transfer.

Since im2col, which must be performed before GEMM, replicates input feature map data for GEMM execution, only offloading the GEMM to the accelerator increases the required data storage and transfer, which is not desirable for cognitive IoT systems. Hence, we combine the im2col, GEMM, and accumulation operation within a unified hardware accelerator referred to as 'iMAC'. Fig. 5 shows an overall architecture of our iMAC accelerator. From the input BRAM, input feature map data is delivered to the im2col functional units (FUs) that perform im2col operations for a certain area of the input feature map. Data that is unrolled from the im2col FU is delivered to Im2colBuffer, which is multiplied by the weights from the weight BRAM. Since we immediately calculate the MAC operation with data from im2col functional unit (FU), we do not need to maintain replicated data in the memory, implying that we can reduce the amount of the data transfer and required size of the data storage compared to the case of only offloading the GEMM. The multiplied results are temporarily stored in multiplication buffer (MulBuffer) and then accumulated to output buffer (OutBuffer). Finally, the results in OutBuffer are also added to the Output BRAMs for accumulation across the results from the multiple input channels.

To further improve performance, we can employ multiple

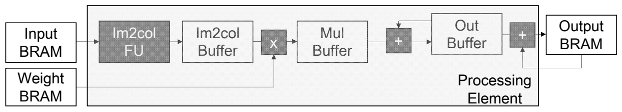

Fig. 5. Hardware architecture of our iMAC (im2col+MAC) accelerator with one processing element (PE). We can also use multiple PEs to increase throughput.

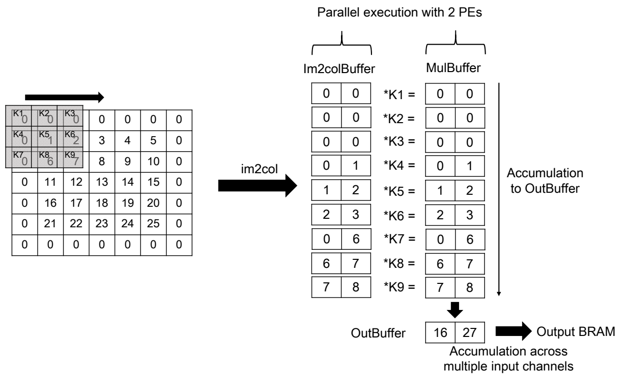
Parallel execution with 2 PEs
Im2colBuffer
MulBuffer
im2col
Accumulation to OutBuffer
OutBuffer
Output BRAM
Accumulation across multiple input channels

Fig. 6. Im2col execution in our iMAC hardware accelerator. In this example, we have 5×5 input feature map with 3×3 filter with 1 zero-padding and 1 stride. Filter weights are assumed to be all 1.0 in this example.

processing elements (PEs) for parallel executions of the multiple im2col and MAC operations. Fig. 6 shows an example execution sequence of our iMAC accelerator with two PEs. In this case, we can simultaneously perform im2col operations for two columns which will be temporarily stored into Im2colBuffer in each PE. We also carry out multiplications with weights (K1∼K9) and accumulations to OutBuffer and output BRAM for two columns (one column is processed with one PE) at the same time, increasing the computation throughput compared to the case of using a single PE.

Our hardware design focuses on cost efficiency. Through fine-grained offloading (we only offload the portions that take a huge latency, e.g., GEMM), we use minimal hardware resources. Thus, our hardware is suitable for accelerating CNNs in resource-constrained IoT systems. Due to the limited on-chip memory (BRAMs), the weight data could not be reused in our design. We may trade the input BRAMs for weight BRAMs in order to increase the weight reusability. However, decreasing the size of input BRAMs has two important disadvantages: (1) it may also reduce the PE utilization due to the reduced input data supply and (2) we need more frequent input data transfer, which results in worse performance and more power consumption.

C. Software Supports

1) Channel Partition and Input/Weight Allocation to Hardware Accelerator: Provisioning the hardware resources for in-situ execution of the entire convolution layer could not be a desirable solution for cognitive IoT systems. Since our design is geared towards resource-constrained systems, we only have limited on-chip memories and processing elements which are not enough to implement a whole convolution layer. Since our iMAC hardware can only execute a part of the convolution layer (i.e., only a part of the entire input channels in a convolution layer) at once, efficient partition and allocation of input channels to the iMAC hardware accelerator are very crucial.

```
Pseudocode for Software-Level Supports
1   out = 0;
2   for (i=1; i<=NumOutputChannel; i++)
3     for (j=1; j<=NumPartition; j++)
4       DMA_Transfer_to_iMAC (jth_Weight);
5       while (!DMA_done);
6       DMA_done = 0;
7       DMA_Transfer_to_iMAC (jth_InputFeatureMap);
8       if (out)
9         Bias_Addition_and_Activation ((i-1)th_OutputChannel);
10        out = 0;
11      endif
12      while ((!Check_iMAC_done()) || (!DMA_done));
13      DMA_done = 0;
14    endfor
15    DMA_Transfer_to_Memory (ith_OutputChannel);
16    out = 1;
17    while (!DMA_done);
18    DMA_done = 0;
19  endfor
20  Bias_Addition_and_Activation ((i-1)th_OutputChannel);
```

Fig. 7. Pseudocode of our software-level supports. The 'DMA_done' is a flag variables that autonomously set as '1' as soon as the DMA transfer is finished. The 'Check_iMAC_done()' is a function that returns whether or not the iMAC accelerator is finished (it returns '1' if finished, '0' if not).

In our design, the hardware accelerator (iMAC) operates by a unit(s) of an input channel. Thus, we try to put the input feature map and weight data for as many input channels as possible to the input and weight BRAMs, respectively, to minimize the number of data transfer. Fig. 7 shows a pseudocode of our channel partition method controlled by software. One outer loop iteration generates results for one output channel (lines 2-19). Thus, if we would like to generate more than one output channel, we need to execute the outer loop by $NumOutputChannel$ times (i.e., equal to the number of the total output channels in a certain convolution layer). For each inner loop iteration (lines 3-14), we generate partial results for one output channel. The inner loop iterations are executed by $NumPartition$ times where $NumPartition$ is equal to ⌈(the total number of input channels / the number of maximum input channels that our iMAC hardware can accommodate)⌉. How many input channels can be accommodated is bound to input and weight BRAM size. For a hypothetical example, we have 50,176×4B input BRAMs with 288×4B weight BRAMs. We also have 112×112×12 input feature maps (an element size=4B: one floating-point variable) with 32 3×3×12 weights (filters). The input and weight BRAM can accommodate input feature maps for up to 4 input channels (50,176*4B=**4***112*112*4B) and 12 layers of filters (288*4B ≥**12**×3×3×4B), respectively. In this example, the number of input channels we can accommodate is bound to the input BRAM size (12>4). Thus, $NumOutputChannel$ and $NumPartition$ values will be set as '32' (same as the number of the filters) and '3' (=12 input channels / 4 channels at maximum with available BRAMs), respectively.

D. Exploiting Parallelism within Convolution Layer Operations

By exploiting the parallelism that exists in different operations in the convolution layer, our design implements a

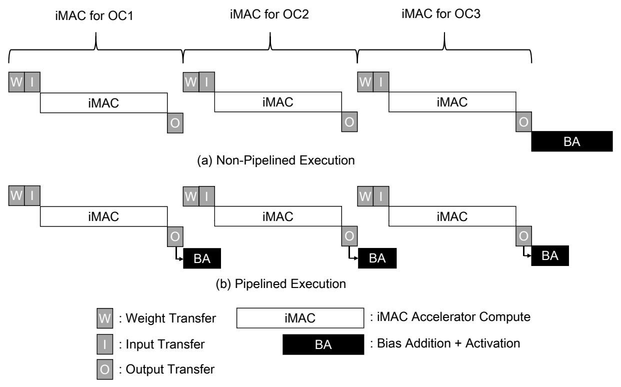

Fig. 8. Comparison between (a) non-pipelined and (b) pipelined execution.

pipelined execution of the operations. As shown in Fig. 7, in lines 7-11, as soon as the DMA transfer is triggered for input feature maps (during the DMA transfer, the CPU can perform another task), the bias addition and activation for the previous output channel are performed (lines 8-11). Since our iMAC hardware is implemented to autonomously perform the im2col and MAC operations as soon as finishing the DMA transfer for input feature maps, the CPU waits for finishing the DMA transfer for input feature map and im2col and MAC operations in the while loop (line 12). In lines 15-18, DMA transfer for the results of output channel i is performed and the CPU waits for finishing the DMA transfer. After the termination of the outer loop iterations, the bias addition and activation operations for the last output channel ($i-1$ because we already incremented i by 1 before we terminate the outer loop) are carried out in the CPU in line 20.

For comparison, Fig. 8 shows the timing diagrams without pipelined execution (a) and with pipelined execution (b) for three output channels (OCs). Without pipelining (Fig. 8 (a)), the bias addition and activation are performed when the iMAC hardware computation and DMA transfers for outputs are entirely finished. Thus, iMAC accelerator and CPU are not used in parallel and either of hardware resources is idle, causing a throughput loss. In our design, as shown in Fig. 8 (b), we exploit the data independence between the output channels. It means we can overlap the executions of the bias addition and activation for the previous output channel $N-1$ with the iMAC execution for the current output channel N. Please note that the bias addition and activation for the last output channel of a convolution layer cannot be overlapped as shown in Fig. 8 (b). This part also corresponds to the line 20 in the pseudocode shown in Fig. 7.

In our design, we do not apply double buffering. The rationale behind this design decision is cost (resource) efficiency. To support double buffering with the same PE utilization, we need twice more BRAMs, which is not desirable for resource-constrained system. In our implementation, the computation time dominates the data transfer time, which also means a performance gain from double buffering would be marginal. Instead of applying the double buffering, we exploit parallelism in the convolution layer operations by overlapping the im2col+GEMM and bias addition/activation, leading to cost- and resource-efficient design.

TABLE I
LOGIC ELEMENT AND BRAM USAGE OF THE DESIGN PRESENTED IN [5], [6], [9], AND OURS. THE NUMBERS OF REGISTERS AND LUTS ARE APPROXIMATE VALUES.

	[9] (Virtex7)	[6] (Zynq-7045)	[5] (Zynq-7045)	Our design (Zynq-7020)
Register	205K	127K	61K	18K
LUT	185K	182K	100K	16K
DSP	2240	780	864	50
36Kb BRAM	512	486	320	133.5

IV. FPGA IMPLEMENTATION

We implement our design in ZED platform [1], which has a Xilinx Zynq 7020 FPGA-SoC that integrates programmable logic and ARM Cortex-A9 CPU. For iMAC implementation, we use Vivado HLS (High-Level Synthesis) that translates high-level programming language to Zynq-compatible hardware design. In order to actually implement CNN, we use Darknet [2] framework with PetaLinux. We also modified the software code of Darknet to support our optimizations. For our CNN model, we use Tiny-Darknet [3][1]. Our implementation is based on 32-bit floating-point CNN model which can be more flexibly applied to many types of different CNN models and frameworks. The FPGA clock frequency is set to 90MHz. Table I shows the hardware utilization of our implementation for iMAC accelerator with 8 PEs compared to the existing designs [5] [6] [9] that use Xilinx FPGA-based platforms. Since our design is geared towards resource-constrained system, we use much less logic and memory elements in Zynq FPGA-SoCs. In terms of BRAM, we only use around 600KB, which is also less than the other implementations shown in Table I. Though our design additionally uses 728 LUTRAMs (not shown in Table I), the hardware cost of an LUTRAM is comparable to an 64-bit storage, meaning that overhead of the additional LUTRAMs is negligible ($\sim$5.8KB). Compared to the existing designs, our design is more suitable for resource-constrained cognitive IoT where we have much lower budget for power and hardware resource.

V. EXPERIMENTAL RESULTS

We show our experimental results with four different designs: *CPU*, *GEMM*, *iMAC_NPL*, and *iMAC_PL*. *CPU* only uses CPU without an additional hardware accelerator. *GEMM* only offload GEMM to the dedicated FPGA-based logic. *iMAC_NPL* offloads im2col and MAC to the dedicated FPGA hardware (iMAC accelerator) while the pipelining described in Section III-D is not applied. *iMAC_PL* offloads im2col and MAC to the iMAC accelerator with the pipelining supports in the software. We also show our results when the SIMD (ARM NEON) support, which can be orthogonally applied to four different designs, is provided and not provided in the CPU. In the cases of *iMAC_PL* and *iMAC_NPL*, the identical 8PE-based iMAC hardware accelerator is used. For fair comparison, we use the same amount of BRAM blocks in *GEMM* (with 95MHz FPGA clock) as that in *iMAC_PL*.

Fig. 9 shows our response time and energy results for eight cases (4 different designs * 2 (w/ NEON and w/o NEON)).

[1]Please note that our implementation is not limited to Tiny-Darknet and other CNN models can also be implemented with only configuring the network architecture description files in Darknet.

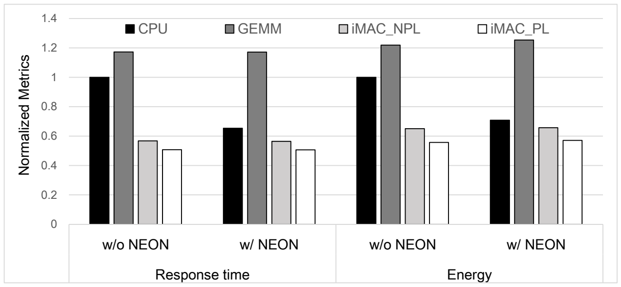

Fig. 9. Response time and energy comparison. The results are normalized to the results of *CPU* w/o NEON.

In the case of response time, our *iMAC_PL* significantly reduces response time of the CNN inference regardless of the NEON support is available or not. When the NEON support is not available, the speedup of our *iMAC_PL* is 1.97× compared to *CPU*. It means our cognitive engine can achieve a huge speedup in resource-constrained embedded/IoT environments with a small hardware overhead. In the case of *GEMM*, response time is rather increased compared to *CPU*. This is because of the huge increase in the amount of data transfer caused by data replication (i.e., unrolling the input) during im2col. Compared to *iMAC_NPL*, *iMAC_PL* shows 1.12× speedup which is attributed to the overlap between the computation times for data transfer/accelerator and bias addition/activation. The relative performance impact of pipelining could be further improved as we put more PEs and/or increase the clock frequency of the iMAC so that we can reduce the execution time of the im2col and MAC operations. When the NEON support is available, performance of *CPU* is increased by 52.9% compared to *CPU* without NEON. However, *iMAC_PL* with NEON still shows better performance by 29.2% compared to *CPU* with NEON. It implies our acceleration technique can achieve better response time regardless of NEON supports in the CPU. In the case of platform-level energy, our *iMAC_PL* shows the highest energy reductions among *GEMM*, *iMAC_NPL*, and *iMAC_PL* compared to *CPU*. Our *iMAC_PL* shows energy reductions by 44.3% and 19.4% compared to the *CPU* without and with NEON supports, respectively.

VI. Related Work

Recently, many FPGA-based CNN acceleration techniques have been proposed. Sugimoto et al. [7] have proposed a method to accelerate an execution of the convolution layer with GEMM hardware though it has a limitation of using software-based im2col which increases the amount of data size. In [6], Qiu et al. have proposed an acceleration technique via dynamic data quantization and convolver design, leading to improving bandwidth and resource utilization in an embedded FPGA platform. Meloni et al. [5] have proposed a new methodology to utilize both hardware accelerator and embedded CPU in Zynq-based FPGA-SoC platform. This methodology utilizes a 16-bit fixed point convolution engine implemented in the programmable arrays and ARM CPU's NEON units for processing the convolution layers. The proposed software framework orchestrates the overall convolution layer processing. Zhong et al. have [10] proposed a unified framework for accelerating CNNs on heterogeneous embedded platforms. The proposed framework utilizes GEMM hardware with multi-threading to efficiently hide latencies. Compared to the FPGA-based works introduced above, our proposed design focuses on cost-efficient implementation (as shown in Table I) with a novel convolution hardware (iMAC) with software-level optimizations to support our design.

VII. Conclusions

In this paper, we have proposed a HW/SW co-design for cost-efficient convolutional neural network (CNN) inference in cognitive Internet of things (IoT). For the cost-efficient design, we only offload the most critical parts in convolution layers (i.e., im2col and GEMM) to the FPGA. We have further proposed software-level supports to efficiently partition and allocate the input channels to our hardware accelerator and exploit the parallelism inside the convolution layer operations. Results demonstrate the hardware resource efficiency of the FPGA-based implementation of our HW/SW co-design. Experimental results reveal that our implementation attains 1.3×∼2.0× speedup and energy reduction of 19.4%∼44.3% as compared to using only the CPU. Results verify that our HW/SW co-design achieves a good trade-off between response time, energy, and cost for CNN inference in cognitive IoT under tight resource budgets.

VIII. Acknowledgement

This research was supported by the Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Education (NRF-2018R1D1A3B07045908).

References

[1] Avnet. Available at: http://www.zedboard.org/.

[2] Darknet: Open Source Neural Networks in C. Available at: https://pjreddie.com/darknet/.

[3] Tiny Darknet. Available at: https://pjreddie.com/darknet/tiny-darknet/.

[4] K. Lee, J. Kong, Y. G. Kim, and S. W. Chung. Memory streaming acceleration for embedded systems with cpu-accelerator cooperative data processing. *Microprocessors and Microsystems - Embedded Hardware Design*, 71, 2019.

[5] P. Meloni, A. Capotondi, G. Deriu, M. Brian, F. Conti, D. Rossi, L. Raffo, and L. Benini. Neuraghe: Exploiting CPU-FPGA synergies for efficient and flexible CNN inference acceleration on zynq socs. *CoRR*, abs/1712.00994, 2017.

[6] J. Qiu, J. Wang, S. Yao, K. Guo, B. Li, E. Zhou, J. Yu, T. Tang, N. Xu, S. Song, Y. Wang, and H. Yang. Going Deeper with Embedded FPGA Platform for Convolutional Neural Network. In *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 26–35, 2016.

[7] N. Sugimoto, T. Mitsuishi, T. Kaneda, C. Tsuruta, R. Sakai, H. Shimura, and H. Amano. Trax solver on Zynq with Deep Q-Network. In *2015 International Conference on Field Programmable Technology (FPT)*, pages 272–275, 2015.

[8] Q. Wu, G. Ding, Y. Xu, S. Feng, Z. Du, J. Wang, and K. Long. Cognitive Internet of Things: A New Paradigm Beyond Connection. *IEEE Internet of Things Journal*, 1(2):129–143, April 2014.

[9] C. Zhang, P. Li, G. Sun, Y. Guan, B. Xiao, and J. Cong. Optimizing FPGA-based Accelerator Design for Deep Convolutional Neural Networks. In *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 161–170, 2015.

[10] G. Zhong, A. Dubey, C. Tan, and T. Mitra. Synergy: A HW/SW Framework for High Throughput CNNs on Embedded Heterogeneous SoC. *CoRR*, abs/1804.00706, 2018.