

Received October 22, 2020, accepted November 13, 2020, date of publication November 19, 2020, date of current version December 7, 2020.

Digital Object Identifier 10.1109/ACCESS.2020.3039278

CPU-Accelerator Co-Scheduling for CNN Acceleration at the Edge

YEONGMIN KIM¹, (Student Member, IEEE), JOONHO KONG², (Member, IEEE), AND ARSLAN MUNIR³, (Senior Member, IEEE)

¹School of Electronic and Electrical Engineering, Kyungpook National University, Daegu 41566, South Korea

²School of Electronics Engineering, Kyungpook National University, Daegu 41566, South Korea

³Department of Computer Science, Kansas State University, Manhattan, KS 66506, USA

Corresponding author: Joonho Kong (joonho.kong@knu.ac.kr)

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).

ABSTRACT Convolutional neural networks (CNNs) are widely deployed for many artificial intelligence (AI) applications, such as object detection and image classification. Due to the burgeoning revolution in edge AI, CNN hardware accelerators are also being employed in resource-constrained edge devices for achieving better performance and energy efficiency at the edge. Although CNN accelerators enable fast and energy-efficient CNN inference at the edge, the remaining hardware resources on the edge devices except for the CNN accelerator remain idle, which could otherwise be utilized for attaining even better performance and energy efficiency for CNN inferences. In this paper, we propose a CPU-accelerator co-scheduling technique for convolution (CONV) layer operations of CNN inferences in resource-constrained edge devices. Our proposed co-scheduling technique exploits an inherent parallelism in CNN output channels, that is, the operations for generating different output channels in a CONV layer can be executed in parallel. For load balancing between the CPU and the CNN accelerator, we also propose a simple, yet accurate latency model for CONV layer operations in the CPU and the accelerator. Based on the latency estimation of CONV layer operations provided by our proposed model, we distribute the tasks to the CPU and the CNN accelerator in a load-balance manner to minimize the idle period during the CONV layer operations in both the CPU and the CNN accelerator. We implement our proposed hardware/software (HW/SW) co-scheduling technique in various field-programmable gate array system-on-chip (FPGA-SoC) platforms as a proof-of-concept. Experimental results indicate that our proposed co-scheduling technique improves system performance by $1.18\times - 2.00\times$ with energy reduction of $14.9\% - 49.7\%$ as compared to the accelerator-only execution.

INDEX TERMS Convolutional neural networks, resource-constrained edge devices, co-scheduling, latency model, load balancing.

I. INTRODUCTION

Recent advancements in artificial intelligence (AI), in particular convolutional neural networks (CNNs) that provide object detection and image classification, have revolutionized a number of real-life applications, such as transportation, agriculture, industrial automation, and home monitoring systems. Furthermore, with proliferation of Internet of things (IoT) devices, billions of IoT devices are connected to the Internet generating zettabytes of data at the network edge. Traditionally, big data, such as online shopping records, social media contents, airlines flight data, and other business-related data, has been stored and analyzed at cloud

The associate editor coordinating the review of this manuscript and approving it for publication was Zhaojun Li.

data centers. However, the recent trend is to push the artificial intelligence (AI) frontiers to the network edge where most of the data is generated thus enabling many novel applications, such as autonomous driving, surveillance, voice assistants, and robots. Figure 1 depicts a framework of edge AI where AI is performed at edge devices in coordination with edge servers [1]. Figure 1 shows that the results and data from edge devices and edge servers are still transferred to cloud for archival and global analytics. Since computing power of edge devices and servers is much less than that of the cloud servers, it calls for fast, yet efficient AI/CNN inference accelerators along with algorithmic advancements and the techniques, such as quantization and network pruning, that enable cost-efficient AI/CNN inference. This paper focuses on CNN inference and acceleration at the edge.

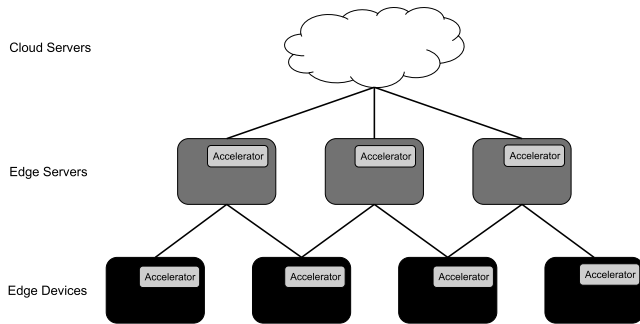


FIGURE 1. A framework for edge AI. The accelerator in the edge devices and servers help enable fast and efficient AI/CNN inference.

Although many hardware accelerators and algorithmic advancements have been proposed for CNNs, efficient CNN inference in resource-constrained edge systems is still an arduous undertaking. Typical system-on-chips (SoCs), which are generally used in many resource-constrained edge systems, employ many different hardware intellectual properties (IPs) to support a wide range of tasks. Recently CNN accelerators are also being incorporated as hardware IP in modern edge devices and servers. For CNN inference in such environments, the main central processing unit (CPU) triggers the CNN accelerator with direct memory access (DMA)-based data transfer. The CNN accelerators in most cases can execute CNN inferences faster than the CPU or other hardware IPs. However, in most of the contemporary CNN acceleration systems, the CPU or hardware IPs other than the CNN accelerator remain idle during the CNN inference. These idle hardware resources could otherwise be utilized for accelerating the CNN inferences alongside the CNN accelerator. For example, a typical CPU in embedded platforms can perform matrix-vector operations. With single instruction multiple data (SIMD) supports, we can make these operations much faster. Consequently, we can utilize the idle hardware resources in CPU to offload a certain portion of CNN inference tasks to make the CNN inference faster than using only the CNN accelerator.

For utilization of hardware resources in cognitive (intelligent) SoC-based platform, an important design decision is how we can distribute the CNN inference tasks in a load-balanced manner. The ‘ideal load-balancing’ for CNN inference does not mean the traditional load balancing where the absolute amount of the tasks are evenly distributed but implies the distribution of task in a manner that maximizes the utilization of multiple hardware resources (i.e., ideal load balancing minimizes the idle time of SoC hardware resources). Hence, there would be load imbalance between various hardware resources of an SoC platform if *relative performance ratio* (i.e., performance of A relative to B = $\frac{Perf_A}{Perf_B}$ where A and B are certain subsystems or components) is not carefully considered. Load imbalance can result in idle time in various hardware resources, which leads to underutilization of the hardware resources. Thus, in order to minimize load imbalance in SoC platforms (i.e., to minimize idle time in hardware resources), a careful consideration of task

distribution is required at a fine granularity by considering the relative performance ratio.

In this paper, we propose a technique to utilize idle hardware resources for convolution (CONV) layer acceleration, which generally constitutes the largest portion of CNN inference latency. Our proposed technique exploits the CPU for sharing load of CNN inference tasks along with the CNN hardware accelerator. The main reasons of utilizing CPUs for sharing CNN inference load with the hardware accelerator in our proposed technique are: (1) CPUs are hardware IPs employed in most of the resource-constrained edge systems, and (2) CPUs can execute matrix-vector (or matrix-matrix) operations with reasonable efficiency [2], which makes CPUs attractive components for sharing CNN tasks with the CNN accelerator. In order to distribute CNN inference tasks between the hardware accelerator and the CPU, we exploit the parallelism in generating multiple CNN output channels (which can be generated independently) in a single CONV layer. To help minimize the idle time in both the accelerator and the CPU, we also propose a linear regression-based latency model for CONV layer execution. By referring to the estimated latency from our model, our proposed co-scheduling technique distributes the CNN output channels between the accelerator and the CPU to maximize the utilization of the both (i.e., to minimize the idle time in both the accelerator and the CPU).

In fact, there have been previous works that achieve more than $10\times$ reduction of the computation [3] or more than $40\times$ weight size reduction [4], [5] with a negligible accuracy loss; however, those works focus on improving the CNN models [5] or changing data format (quantization) and value (pruning) of the weights [3], [4]. On the contrary, our proposed acceleration technique utilizes CPU and the CNN accelerator co-scheduling, and can be employed orthogonally with the existing CNN acceleration techniques that utilize quantization and pruning. Furthermore, our proposed technique is applicable to any CNN model because our technique focuses on utilizing both the CPU and the CNN accelerator when executing a CNN inference with a given CNN model (i.e., our technique is not limited to any particular CNN model). Thus, our proposed technique can be much more broadly applied for CNN inference acceleration at the edge as compared to other CNN acceleration techniques.

We summarize our main contributions in this work as follows:

- We propose a CNN output channel distribution technique with CPU-accelerator co-scheduling to utilize both the CNN accelerator and the CPU;
- We propose a simple, yet accurate convolution layer latency model for the CNN accelerator and the CPU;
- We implement our proposed co-scheduling technique in a CNN framework [6] while verifying it in various field programmable gate array system-on-chip (FPGA-SoC) based platforms as a proof-of-concept;
- The experimental results reveal that our proposed co-scheduling technique improves the performance of

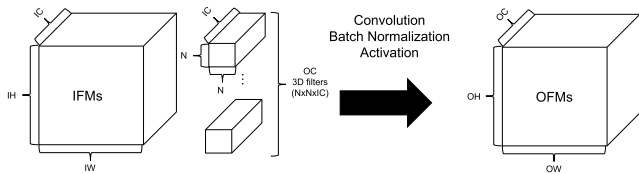


FIGURE 2. Typical convolutional layer operations. IH, IW, IC, OH, OW, and OC stand for input height, input width, input channel, output height, output width, and output channel, respectively.

convolution layer operations by $1.18\times - 2.00\times$ while at the same time reducing the energy consumption by $14.9\% - 49.7\%$ as compared to the accelerator-only execution.

The remainder of this paper is organized as follows. Section II describes background and preliminary information for this work. Section III presents our proposed CPU-accelerator co-scheduling technique. Experimental results based on our implemented prototypes are presented in Section IV. Section V summarizes recent literature related to our work. Lastly, Section VI concludes this paper.

II. BACKGROUND AND PRELIMINARIES

A. CONVOLUTIONAL NEURAL NETWORKS

CNNs are generally composed of multiple different types of layers. Though some CNN architectures contain specialized layers (e.g., Inception layer in [7]), most of the modern CNN architectures are composed of convolutional (CONV) layers, pooling layers, and fully connected layers. Among those layers, it is known that the CONV layer takes the largest portion of the execution time in a CNN inference. Thus, in this paper, we mainly focus on accelerating the CONV layers.

A deep look into the CONV layer (Figure 2) divulges that input feature maps (IFMs) and weights (filters) mainly constitute an input for the CONV layer. Bias values (*gamma* and *beta* for batch normalization) are sometimes included in the inputs for CONV layers though they are omitted in Figure 2. With the IFMs ($IH \times IW \times IC$) and weights ($N \times N \times IC \times OC$), the convolution operations are performed. After the convolution, depending on the CNN models, the batch normalization can be performed to the output from the convolution operations. Lastly, the activation is performed, typically by rectified linear units (ReLUs), which generate the output feature maps (OFMs) of dimension $OH \times OW \times OC$.

B. BASELINE SYSTEM ARCHITECTURE

In this work, we assume a resource-constrained edge system with a CNN accelerator (such as Coral platform [8] which has a CPU with edge tensor processing unit). As shown in Figure 3, our baseline SoC is similar to a typical embedded system which includes CPU and memory controller. In our baseline system, we also use the CNN accelerator to expedite the CNN inference in the system. Though there can be various types of CNN accelerator, we assume that the CNN accelerator leverages multiple processing elements (PEs) by exploiting the parallelism in the output channels of the OFMs. It means that each PE generates different 2D ($OH \times OW$) output feature maps (OFMs). For example, if we have M PEs

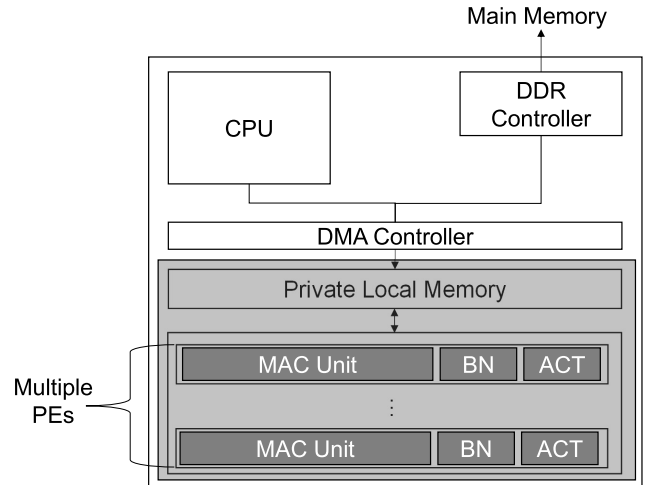


FIGURE 3. Baseline system architecture. The light gray-shaded area corresponds to the CNN accelerator while the dark gray-shaded area corresponds to the processing elements in the accelerator.

in the accelerator, we can generate M output channels simultaneously. Please note that this type of accelerator design which exploits output channel parallelism for multiple PE design is widely used [9], [10] [11]. Inside the PE, there are multiply-and-accumulation (MAC) units, batch normalization (BN) units, and activation (ACT) units. The accelerator on-chip memory, which is also referred to as private local memory (PLM) is used as a local buffer in which we store IFMs, weights, and OFMs (or intermediate results). Similar to typical embedded systems, we use DMA for data transfer between the CPU's main memory and the accelerator's PLM.

III. CNN ACCELERATION WITH CPU-ACCELERATOR CO-SCHEDULING

A. OVERVIEW

To fully utilize both the CNN accelerator and the CPU in resource-constrained edge devices, this work aims at distributing the CNN CONV layer tasks to the CNN accelerator and the CPU in a load-balanced manner by carefully considering the performance ratio between the CPU and the accelerator. Since there is no dependency when generating different output channels (i.e., they can be executed in parallel) in a single CONV layer, we divide the output channels to distribute the tasks into the accelerator and CPU. Consequently, we perform convolution layer operations in the accelerator and the CPU with different 3D filter tensors. Figure 4 demonstrates an overview of our co-scheduling technique. To fully utilize the accelerator and CPU, we need to estimate the relative difference of the achievable performance in the accelerator and CPU. For accurate estimation, we use a latency model of the CONV layer operations for the accelerator and CPU, which will be described in the following subsection in details. With the estimated relative attainable performance of the accelerator and CPU, we distribute output channels (i.e., 3D filter tensors) to the accelerator and CPU. The information on the output channel distribution is stored in the CNN framework (Figure 4), which will be referenced at

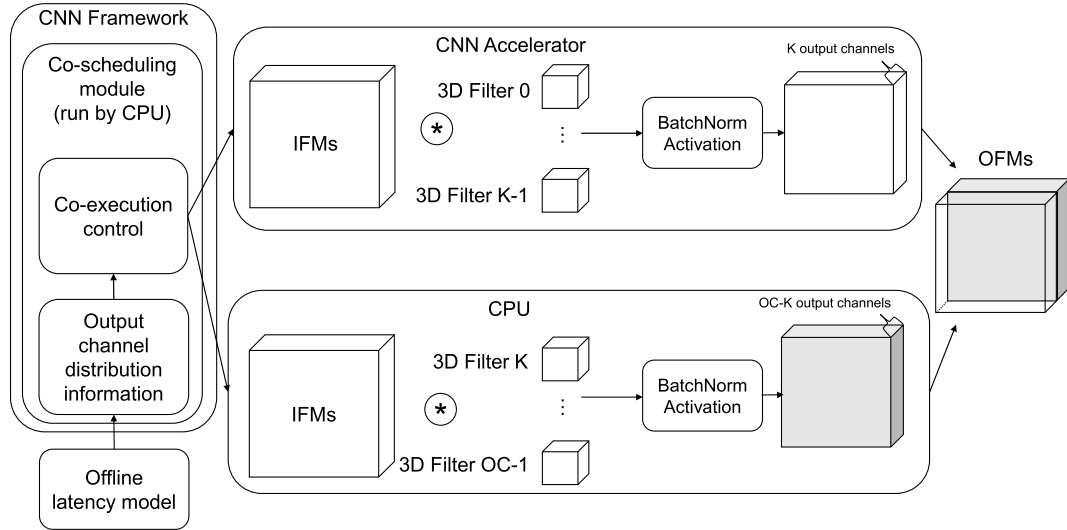


FIGURE 4. An overview of our proposed CPU-accelerator co-scheduling.

runtime. In the example shown in Figure 4, we assign “ K ” 3D filter tensors (which will be used for generating “ K ” output channels) to the CNN accelerator while we assign “ $OC - K$ ” filter tensors to the CPU. At runtime, the co-execution control module, which is executed on the CPU in our co-scheduling technique, launches the accelerator and CPU co-execution for given output channels. The CONV layer operation includes convolution, batch normalization (BatchNorm), and activation. After we generate “ K ” and “ $OC - K$ ” 2D output feature maps (OFMs) in the accelerator and the CPU, respectively, we aggregate those OFMs to compose the full 3D OFM tensor.

B. LINEAR REGRESSION-BASED LATENCY MODEL

For latency estimation, we use a linear regression-based methodology. In general linear regression, we use a following form of the equation:

$$Y = \alpha X + \beta \quad (1)$$

where X and Y are explanatory and dependent variables, respectively. With the pairs of X and Y values, we determine α and β values through the linear regression training (line fitting). Based on the above form of the equation, we extend it to estimate accelerator and CPU latencies when processing a single CONV layer.

1) ACCELERATOR LATENCY MODEL

In this paper, we propose a linear regression-based latency model for CONV layer executions in the CNN accelerators introduced in Section II-B. For typical task offloading to the hardware IPs in embedded systems, we firstly need to send the data from the main memory to the accelerator’s PLM via DMA. After that, the accelerator executes the offloaded computations and send the result data back to the main memory via DMA. Between the data transfer, if we do not use a cache-coherent interconnect, we also need to perform CPU data cache flush and invalidation for software managed cache coherence, which also takes non-negligible latency

depending on the amount of the data to be transferred. Thus, an accelerator latency requires to be broken down into three different parts: computation, data transfer, and coherence latency.

a: COMPUTATION LATENCY

For computation latency, it mainly depends on how many operations we need to generate the OFMs. Firstly, we generate a unit latency L_{uACC} that corresponds to the computation latency in the 1PE accelerator (i.e., the accelerator with one PE) when generating one OFM element. Deriving L_{uACC} can be done as follows:

$$L_{uACC} = \alpha_{comp} \times Size_{FT} + \beta_{comp} \quad (2)$$

where $Size_{FT}$ indicates the number of elements in one 3D filter tensor. For example, if one 3D filter tensor size is $3 \times 3 \times 3$, the $Size_{FT}$ is 27. The α_{comp} and β_{comp} are determined by the linear regression analysis. We can extend it to derive the accelerator computation latency (L_{comp}) when generating N_F OFMs (i.e., N_F output channels) as follows:

$$L_{comp} = L_{uACC} \times Size_{OFM} \times \left\lceil \frac{N_F}{N_{PE}} \right\rceil \quad (3)$$

where $Size_{OFM}$, N_F , and N_{PE} correspond to the number of the elements in one 2D OFM, the total number of output channels in a certain CONV layer, and the number of PEs in the accelerator, respectively. The $\left\lceil \frac{N_F}{N_{PE}} \right\rceil$ equals to how many times the accelerator execution must be triggered. For example, if we run a CONV layer with $N_F=28$ in the accelerator with $N_{PE}=10$, the number of times the accelerator’s PE execution will be triggered is equal to 3 ($=\lceil 28/10 \rceil$). Accordingly, we calculate the latency of the accelerator when generating the total number of the output channels (i.e., N_F output channels) in a certain CONV layer because we need to calculate the relative performance ratio between the accelerator and CPU. In other words, we should derive the latency of the accelerator and CPU when we execute the CONV layer with the identical input data size.

b: TRANSFER LATENCY

The transfer latency is also linearly proportional to the size of data to be transferred. Thus, the transfer latency can be estimated by the following equation:

$$L_{tran} = \alpha_{tran} \times Size_{Data} + \beta_{tran} \quad (4)$$

where $Size_{Data}$ indicates a total size of the data (in the number of the elements) to be transferred. The α_{tran} and β_{tran} are also determined by the linear regression analysis. The $Size_{Data}$ can be calculated as follows:

$$Size_{Data} = Size_{IFMs} + Size_{gamma} + Size_{beta} + Size_{FT} \times N_F + Size_{OFM} \times N_F. \quad (5)$$

For input data transfer, we need to count $Size_{IFMs}$, $Size_{gamma}$, and $Size_{beta}$ which correspond to the number of elements of (multiple) input feature maps, $gamma$ values, and $beta$ values for batch normalization. Depending on the CONV layer configurations, $Size_{gamma}$ and/or $Size_{beta}$ can be neglected. We also need to count the total number of the elements in N_F 3D filter tensors, which can be derived by multiplying $Size_{FT}$ by N_F . For output transfer, we count the data size (in the number of elements) of the OFMs generated by the accelerator, which is calculated by $Size_{OFM} \times N_F$.

c: COHERENCE LATENCY

The coherence latency model is composed of two parts: cache flush and cache invalidation. The cache flush is required before we trigger DMA read (send input data to the accelerator PLM) while the cache invalidation is required before we trigger DMA write (send output data from the PLM to the main memory). For cache flush latency (L_{fl}), we use the following equation:

$$L_{fl} = \alpha_{fl} \times (Size_{IFMs} + Size_{gamma} + Size_{beta} + Size_{FT} \times N_F) + \beta_{fl} \quad (6)$$

where α_{fl} and β_{fl} correspond to the coefficient values for cache flush. Cache invalidation latency (L_{inv}) can also be estimated by using the following equation:

$$L_{inv} = \alpha_{inv} \times Size_{OFM} \times N_F + \beta_{inv} \quad (7)$$

where α_{inv} and β_{inv} correspond to the coefficient values for cache invalidation. By adding L_{fl} and L_{inv} , we can estimate the total latency required for cache coherence as shown in the following equation:

$$L_{cohr} = L_{fl} + L_{inv}. \quad (8)$$

d: LATENCY AGGREGATION

The total latency taken by the accelerator can be derived by adding the computation, transfer, and coherence latency as follows:

$$L_{ACC} = L_{comp} + L_{tran} + L_{cohr}. \quad (9)$$

In our baseline system, the data transfer, computation, and coherence operations are not overlapped. However, our latency model can also be extended to other systems where an overlap exists between data transfer, computation, and coherence operations. For example, when we use cache-coherent

interconnects, we can just remove L_{cohr} because the explicit cache flush and invalidation are not required in the system with cache-coherent interconnects. When using double buffering where the transfer latency and computation latency can be overlapped, we can replace $L_{comp} + L_{tran}$ with $MAX(L_{comp}, L_{tran})$.

2) CPU LATENCY MODEL

We also propose a linear regression-based latency model for CONV layer executions in the CPU. Different from the accelerator latency model, CPUs use cache memories for data transfer between the CPU core and main memory. Since it is not explicitly controlled by software programmer, accurately estimating the data transfer time in the CPU would be more challenging than that in the accelerator. Hence, we use a unified latency model, which means we do not distinguish the data transfer and computation latency. By leveraging the linear regression-based latency model, we figure out the CPU latency depending on the number of the OFM elements generated by the CPU along with the CPU unit latency (L_{uCPU} : a CPU latency for generating one OFM element). This is because the computation and data transfer (implicitly via caches) latencies will be linearly proportional to the number of the generated OFM elements. The CPU unit latency can be derived as follows:

$$L_{uCPU} = \alpha_{CPU} \times Size_{FT} + \beta_{CPU} \quad (10)$$

where the $Size_{FT}$ equals to the size (in the number of elements) of the one 3D filter tensor. The coefficients α_{CPU} and β_{CPU} , which will inherently incorporate the effect of both computation and data transfer on CPU unit latency, are also determined by the linear regression. With the CPU unit latency model, we can extend it to estimate the total CONV layer latency in the CPU (L_{CPU}) as follows:

$$L_{CPU} = L_{uCPU} \times Size_{OFM} \times N_F \quad (11)$$

where $Size_{OFM} \times N_F$ corresponds to the number of the total OFM elements in a certain CONV layer.

C. CHANNEL DISTRIBUTION

With the estimated latency from our model, we distribute the output channels in order to fully utilize the accelerator and CPU in the system. To accomplish it, we need to distribute the output channels so that the execution time of the accelerator and the CPU is (almost) equivalent. Thus, we use the following equations for the output channel distribution for the accelerator:

$$N_{FA} = \left\lceil \left(\frac{L_{CPU}}{L_{ACC} + L_{CPU}} \right) \times N_F \right\rceil \quad (12)$$

where N_{FA} means the number of the output channels which will be processed in the accelerator. For example, assuming that the accelerator performance is three times better than the CPU (i.e., $L_{ACC}:L_{CPU}=1:3$), the accelerator should process three times more tasks than the CPU in order to fully remove the idle time in both the CPU and the accelerator.

TABLE 1. CPU specifications for four FPGA-SoC platforms.

Platform	Ultra96	Zed	ZCU104, ZCU106
CPU architecture	Quad-core Cortex-A53	Dual-core Cortex-A9	Quad-core Cortex-A53
Clock Frequency	Up to 1200MHz	Up to 667MHz	Up to 1334MHz
Per core L1 Cache	32KB IS, 32KB DS	32KB IS, 32KB DS	32KB IS, 32KB DS
L2 Cache	Shared 1MB	Shared 512KB	Shared 1MB

Obviously, the rest of the filter tensors must be processed in the CPU, which means:

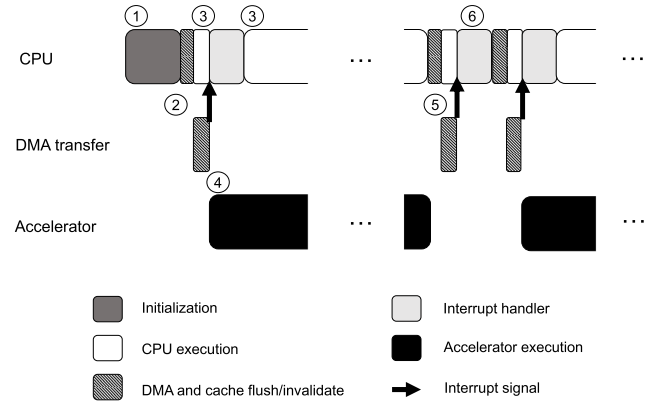
$$N_{FC} = N_F - N_{FA} \quad (13)$$

where N_{FC} corresponds to the number of the output channels which will be processed in the CPU. As we mentioned in Section III-A, determining the N_{FA} and N_{FC} is performed offline, which means the distribution decision for a given embedded system and CNN model is performed at the design time. Once we determine the output channel distribution, this information is stored in the CNN framework (Figure 4). At runtime, when processing a certain CONV layer, the CNN framework can distribute the tasks (output channels) to the accelerator and CPU by the distribution information stored in the framework.

D. PROTOTYPE IMPLEMENTATION

We have implemented our co-scheduling technique in Darknet framework [6], which is used as our baseline CNN framework, and have verified the proposed co-scheduling technique using four FPGA-SoC platforms: Ultra96 [12], Zed [13], ZCU104 [14], and ZCU106 [15]. These platforms deploy various types of the CPUs, which are summarized in Table 1. For our implementation, we use interrupt-based mechanism to simultaneously execute the CONV layer operations on the accelerator and the CPU, which is similar to that introduced in [16]. In our prototype implementation, though we do not employ double buffering, our latency model and co-scheduling can also be extended to support double buffering as we explained in Section III-B1.d. Since our prototype is mainly for verification and proof of the implementation, we run the modified framework in each platform without running an operating system (OS). Please note that this environment is similar to resource-constrained embedded edge devices where the firmware orchestrates the system without running OS. The precision for CNN models used in our work is 32-bit floating-point; however, our technique can also be applied to other precisions such as 16-bit fixed-point and 8-bit integer without modifying the latency model and channel distribution mechanism.

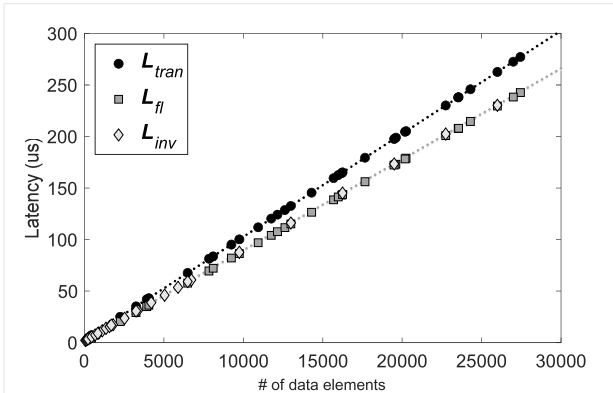
Figure 5 shows the timing diagram of our co-scheduling technique based on the interrupt mechanism. The main framework runs on the CPU and initialization is performed before the co-execution in the CPU and the accelerator. The initialization (①) includes information fetching required for our co-scheduling technique (e.g., N_{FA} and N_{FC}). When a CONV layer execution begins, the CPU launches data transfer which includes cache flush in the CPU and DMA transfer (②). During the DMA transfer, the CPU begins to generate the N_{FC} OFMs (③). As soon as the DMA transfer finishes, the accelerator begins to generate N_{FA} OFMs (④). When the

**FIGURE 5. A timing diagram of our co-scheduling implementation.****TABLE 2. Obtained α and β values through linear regression analysis.**

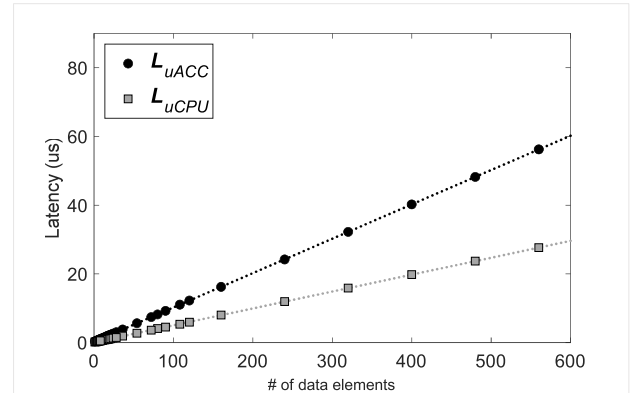
	Ultra96	Zed	ZCU104	ZCU106
α_{tran}	0.01	0.009999	0.009998	0.009999
β_{tran}	2.697551	2.162106	2.458088	2.472711
α_{comp}	0.099999	0.150077	0.090014	0.090021
β_{comp}	0.237558	0.309062	0.218302	0.218306
α_{fl}	0.008811	0.009323	0.006082	0.006101
β_{fl}	0.514771	6.101897	0.491329	0.314022
α_{inv}	0.008812	0.003748	0.006084	0.006091
β_{inv}	1.663402	0.986305	1.036251	0.900236
α_{CPU}	0.049176	0.072254	0.049155	0.049176
β_{CPU}	0.116896	0.164479	0.113563	0.113043

accelerator finishes the execution, a cache invalidation in the CPU and DMA write operation (⑤) begins in order to send the generated OFMs from the accelerator's PLM to the main memory to make the data visible to the CPU. During the DMA write operations, the CPU can continue executing the CONV layer operations for generating the N_{FC} OFMs because the DMA transfer and CPU execution can be performed in parallel. When the accelerator needs to be called for multiple times during a single CONV layer execution because of the larger number of OFMs than the accelerator's PEs can handle concurrently, these steps can be iterated (⑥) until the accelerator and CPU generates N_{FA} and N_{FC} OFMs, respectively.

For prototype implementation, we acquire α and β values via linear regression analysis. For training the linear regression model (i.e., acquiring α and β values), we use synthetically generated arbitrary 32 data points each for regression training of transfer time, flush and invalidation (coherence) time, and computation time. Please note that the range of the arbitrarily generated data points sufficiently covers the range of the data points used in real CNN models, resulting in accurate latency estimation. We demonstrate the coefficient values and plots obtained from our linear regression analysis in Table 2 and Figures 6 – 9, respectively. We use MATLAB for linear regression analysis. As shown in Figures 6 – 9, the computation time and transfer time are well fitted to the linear function (as a form of $Y = \alpha X + \beta$). We can notice

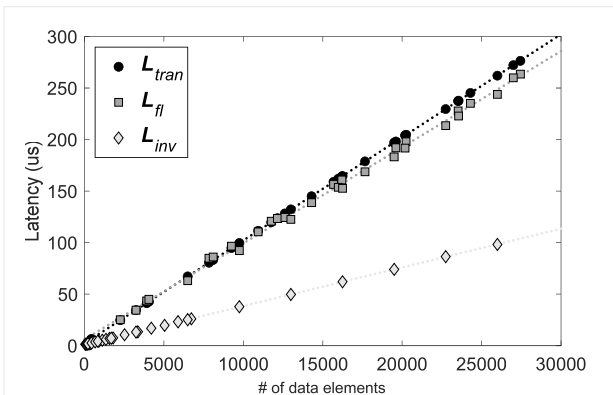


(a) Linear regression for transfer, cache flush, and invalidation latencies

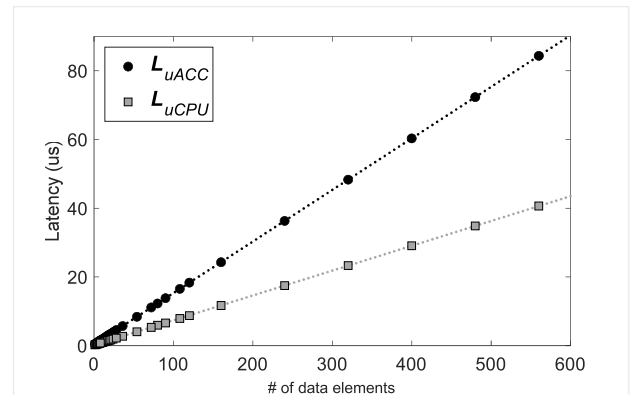


(b) Linear regression for accelerator and CPU unit latencies

FIGURE 6. Plots for linear regressions in Ultra96 platform.

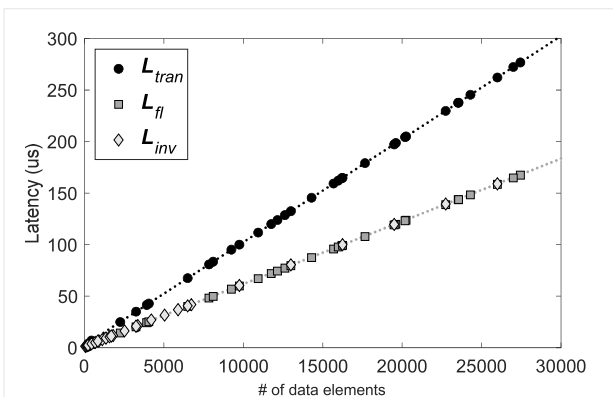


(a) Linear regression for transfer, cache flush, and invalidation latencies

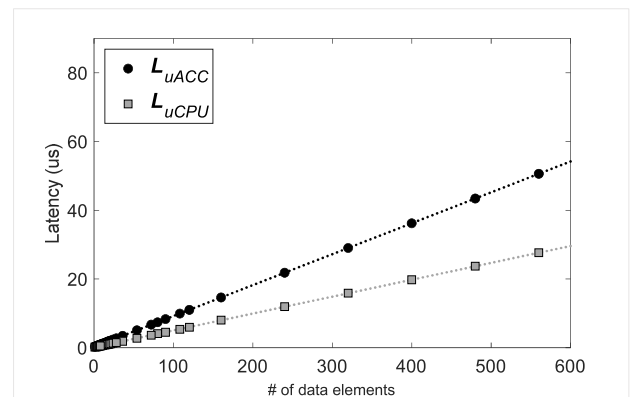


(b) Linear regression for accelerator and CPU unit latencies

FIGURE 7. Plots for linear regressions in Zed platform.



(a) Linear regression for transfer, cache flush, and invalidation latencies



(b) Linear regression for accelerator and CPU unit latencies

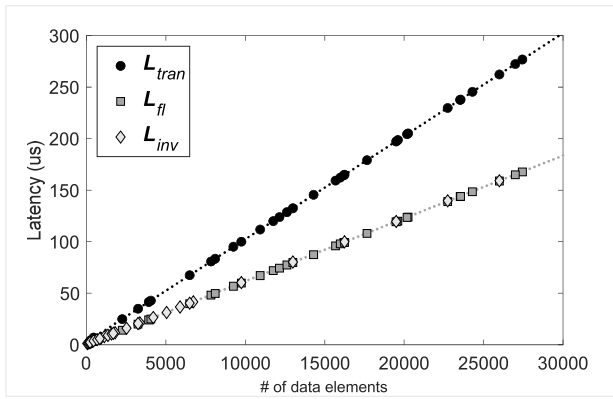
FIGURE 8. Plots for linear regressions in ZCU104 platform.

that the plots in Figures 8 and 9 seem to be almost identical. This is because ZCU104 and ZCU106 have almost identical specifications except for the connectivity parts.

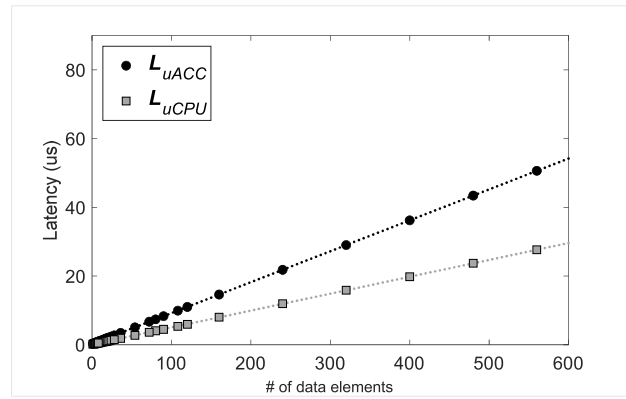
IV. EXPERIMENTAL RESULTS

In this section, we present the experimental results for our proposed co-scheduling technique related to various metrics, such as latency model accuracy (measured via mean absolute

percentage error (MAPE)), performance, and energy consumption. We evaluate these metrics for our implemented prototypes in Ultra96, Zed, ZCU104, and ZCU106. For CPU clock frequencies, we use the following clock frequencies for each platform: Ultra96, ZCU104, and ZCU106 at 1.2GHz and Zed at 667MHz. We also perform the linear regression analysis with these CPU clock frequencies. To evaluate our technique across the wide spectrum of accelerators,



(a) Linear regression for transfer, cache flush, and invalidation latencies



(b) Linear regression for accelerator and CPU unit latencies

FIGURE 9. Plots for linear regressions in ZCU106 platform.

TABLE 3. Convolutional layer configurations for our benchmarks. In the fifth column, SN, MN, and SG indicate SqueezeNet, MobileNet-V2, and Synthetically Generated, respectively.

	$Size_{IFMs}$	$Size_{OFM} \times N_F$	$Size_{FT} \times N_F$	Network
Layer 0	$57 \times 57 \times 16$	$57 \times 57 \times 64$	$1 \times 1 \times 16 \times 64$	SN
Layer 1	$57 \times 57 \times 16$	$57 \times 57 \times 64$	$3 \times 3 \times 16 \times 64$	SN
Layer 2	$29 \times 29 \times 32$	$29 \times 29 \times 128$	$1 \times 1 \times 32 \times 128$	SN
Layer 3	$29 \times 29 \times 32$	$29 \times 29 \times 128$	$3 \times 3 \times 32 \times 128$	SN
Layer 4	$28 \times 28 \times 32$	$28 \times 28 \times 192$	$1 \times 1 \times 32 \times 192$	MN
Layer 5	$15 \times 15 \times 64$	$15 \times 15 \times 256$	$3 \times 3 \times 64 \times 256$	SN
Layer 6	$15 \times 15 \times 48$	$15 \times 15 \times 192$	$3 \times 3 \times 48 \times 192$	SN
Layer 7	$14 \times 14 \times 96$	$14 \times 14 \times 576$	$1 \times 1 \times 96 \times 576$	MN
Layer 8	$7 \times 7 \times 576$	$7 \times 7 \times 160$	$1 \times 1 \times 576 \times 160$	MN
Layer 9	$7 \times 7 \times 320$	$7 \times 7 \times 1280$	$1 \times 1 \times 320 \times 1280$	MN
Layer 10	$7 \times 7 \times 160$	$7 \times 7 \times 960$	$1 \times 1 \times 160 \times 960$	MN
Layer 11	$7 \times 7 \times 160$	$7 \times 7 \times 160$	$3 \times 3 \times 160 \times 160$	SG
Layer 12	$7 \times 7 \times 160$	$7 \times 7 \times 960$	$3 \times 3 \times 160 \times 960$	SG
Layer 13	$7 \times 7 \times 160$	$7 \times 7 \times 1280$	$3 \times 3 \times 160 \times 1280$	SG

we implement three different accelerator versions for each platform while varying the number of PEs: 2PE, 4PE, and 8PE, where xPE denotes an accelerator with “x” number of PEs. Since our target is resource-constrained edge devices, we do not consider the CNN accelerator design with massive number of PEs (e.g., over 100 PEs).

For workloads, we use 14 different CONV layers as shown in Table 3. Six layers (Layer 0, 1, 2, 3, 5, and 6) are from SqueezeNet [5] while five layers (Layer 4, 7, 8, 9, and 10) are from MobileNet-v2 [17]. The rest of the three layers (Layer 11, 12, and 13) are synthetically generated for our evaluation. Please note that the results shown in this section are the averaged results of the 14 CONV layer executions (i.e., we measure the results of each CONV layer separately and average them out).

To further demonstrate the performance improvement of our proposed co-scheduling technique (CPU+ACC), we compare the performance (i.e., per-layer latency) of our proposed technique with the acceleration techniques that utilize only the CPU (CPU_ONLY) and only the accelerator (ACC_ONLY) for all layers of the Tiny Darknet CNN model [18] as a case study using inputs from ImageNet dataset [19].

A. LATENCY MODEL ACCURACY

We show the MAPE results of our latency model. With the obtained α and β values in our latency model, we measured

TABLE 4. MAPE results of our proposed latency model. PE2, PE4, and PE8 corresponds to the MAPEs of the accelerator latency model.

		Ultra96	Zed	ZCU104	ZCU106
L_{ACC} PE2	1×1 CONV	1.01%	0.82%	1.06%	1.06%
	3×3 CONV	0.14%	0.18%	0.13%	0.13%
L_{ACC} PE4	1×1 CONV	0.74%	0.26%	0.63%	0.63%
	3×3 CONV	0.16%	0.17%	0.12%	0.11%
L_{ACC} PE8	1×1 CONV	0.47%	0.98%	0.87%	0.87%
	3×3 CONV	0.26%	0.26%	0.19%	0.19%
L_{CPU}	1×1 CONV	0.65%	0.40%	0.48%	0.47%
	3×3 CONV	0.76%	0.67%	0.68%	0.68%

the error rate between the measured and estimated (from our model) latency of the accelerator and CPU. As summarized in Table 4, the MAPE of our latency model is 0.11% – 1.06%, which means our latency model estimates the accelerator and CPU latency of the CONV layer execution very accurately. It also implies that with the accurate latency estimation, we can distribute output channels to the accelerator and the CPU so that both the hardware resources can be fully utilized, minimizing the idle time and improving performance.

B. PERFORMANCE

We evaluate the performance of our co-scheduling technique through experimental results. Figure 10 shows the performance results across three different configurations: CPU_ONLY (only CPU execution), ACC_ONLY (only accelerator execution), and CPU+ACC (co-scheduling). Experimental results indicate that when we use 2PE accelerator version, relative performance of the accelerator compared to CPU_ONLY is $0.97 \times - 1.09 \times$. This limited performance improvement for 2PE case is due to small number of PEs in the accelerator. Our co-scheduling (CPU+ACC) results in better performance by $1.93 \times - 2.05 \times$ and $1.89 \times - 2.00 \times$ than CPU_ONLY and ACC_ONLY, respectively. By utilizing both accelerator and CPU, our co-scheduling technique leads to better performance than CPU_ONLY and ACC_ONLY.

Experimental results indicate that in the case of 4PE accelerator version, relative performance of the accelerator compared to the CPU is better than the case of 2PE accelerator. Thus, CPU+ACC leads to better performance as compared to the CPU_ONLY by $2.84 \times - 3.07 \times$. As compared to ACC_ONLY, CPU+ACC shows better performance by

TABLE 5. Results for a ratio (%) of the idle time to the total execution time. The bold cells represent the CPU idle time (i.e., the case in which the CPU finishes faster than the accelerator) while the underlined cells denote the accelerator idle time (i.e., the case in which the accelerator finishes faster than the CPU).

CNN CONV Layers	Ultra96			Zed			ZCU104			ZCU106		
	PE2	PE4	PE8	PE2	PE4	PE8	PE2	PE4	PE8	PE2	PE4	PE8
Layer 0	<u>2.22</u>	3.04	6.85	<u>2.81</u>	2.79	7.16	<u>0.14</u>	<u>1.63</u>	6.00	<u>0.13</u>	<u>1.64</u>	5.97
Layer 1	0.35	3.19	1.94	1.02	3.98	3.06	0.85	0.31	0.39	0.84	0.33	0.39
Layer 2	0.63	<u>0.06</u>	<u>1.83</u>	0.41	0.83	6.55	<u>1.34</u>	0.91	3.84	<u>1.33</u>	0.92	3.86
Layer 3	0.73	0.77	1.14	0.60	0.38	2.15	1.02	0.37	1.59	1.02	0.38	1.59
Layer 4	1.11	1.06	<u>1.93</u>	1.50	2.50	<u>0.81</u>	0.61	1.86	2.88	0.61	1.86	2.90
Layer 5	<u>0.07</u>	<u>0.43</u>	<u>1.71</u>	0.31	<u>0.03</u>	<u>0.47</u>	<u>0.33</u>	<u>0.74</u>	<u>1.49</u>	<u>0.33</u>	<u>0.75</u>	<u>1.49</u>
Layer 6	<u>0.66</u>	<u>1.51</u>	<u>1.91</u>	<u>0.25</u>	<u>0.65</u>	<u>0.70</u>	<u>0.74</u>	<u>0.81</u>	<u>0.94</u>	<u>0.74</u>	<u>0.79</u>	<u>0.98</u>
Layer 7	2.91	<u>4.92</u>	<u>2.08</u>	<u>4.18</u>	<u>3.43</u>	<u>0.70</u>	<u>4.14</u>	2.29	<u>4.89</u>	<u>4.10</u>	2.29	<u>4.91</u>
Layer 8	<u>1.17</u>	1.51	<u>2.89</u>	0.17	0.60	<u>1.79</u>	0.56	2.41	1.51	0.57	2.40	1.52
Layer 9	0.73	<u>0.15</u>	<u>0.07</u>	1.85	0.56	<u>0.18</u>	<u>0.45</u>	<u>1.29</u>	1.04	<u>0.45</u>	<u>1.29</u>	1.03
Layer 10	0.08	<u>1.16</u>	1.41	0.86	<u>0.91</u>	0.67	0.82	<u>2.36</u>	<u>1.32</u>	0.82	<u>2.35</u>	<u>1.34</u>
Layer 11	<u>1.36</u>	<u>2.52</u>	<u>4.82</u>	<u>0.90</u>	<u>1.12</u>	<u>6.59</u>	<u>1.95</u>	<u>1.97</u>	<u>3.28</u>	<u>1.94</u>	<u>1.95</u>	<u>3.29</u>
Layer 12	<u>0.82</u>	<u>1.22</u>	<u>1.56</u>	<u>0.23</u>	<u>0.71</u>	<u>1.77</u>	<u>1.01</u>	<u>1.55</u>	<u>1.40</u>	<u>1.00</u>	<u>1.54</u>	<u>1.40</u>
Layer 13	<u>0.85</u>	<u>1.09</u>	<u>2.29</u>	<u>0.44</u>	<u>0.88</u>	<u>1.28</u>	<u>0.61</u>	<u>1.10</u>	<u>1.67</u>	<u>0.60</u>	<u>1.09</u>	<u>1.66</u>

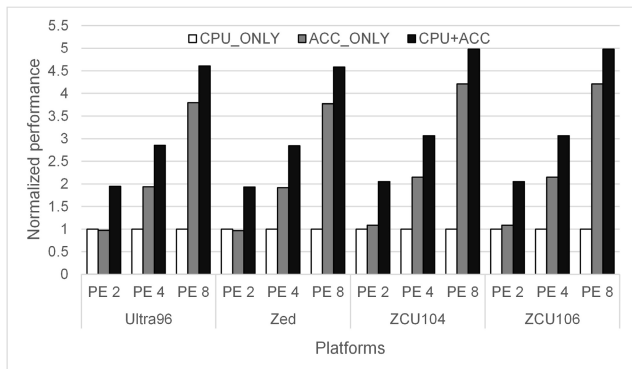


FIGURE 10. Performance results of ACC_ONLY and CPU+ACC normalized to CPU_ONLY.

1.43x–1.48x. Though the relative performance improvement of the CPU+ACC is less than that in the case of 2PE accelerator version, our CPU+ACC still results in better performance due to the concurrent execution of the CONV layer in the accelerator and CPU.

In the case of 8PE accelerator version, the CPU+ACC shows better performance than CPU_ONLY and ACC_ONLY by 4.58x–4.98x and 1.18x–1.21x, respectively. As demonstrated in our evaluation results, due to the accurate latency model, our technique minimizes the idle time in either accelerator or CPU, leading to better performance as compared to CPU_ONLY and ACC_ONLY.

The main reason why our co-scheduling techniques obtains a huge performance improvement is load balancing between the accelerator and CPU. To measure how well our technique distributes the tasks in a load-balanced manner, we present a ratio of the idle time to the total execution time in either the accelerator or the CPU in Table 5. As shown in the results, the average idle time is only 1.61% (maximum idle time is 7.16%), which implies our technique almost completely removes the idle time. Results also demonstrate that the

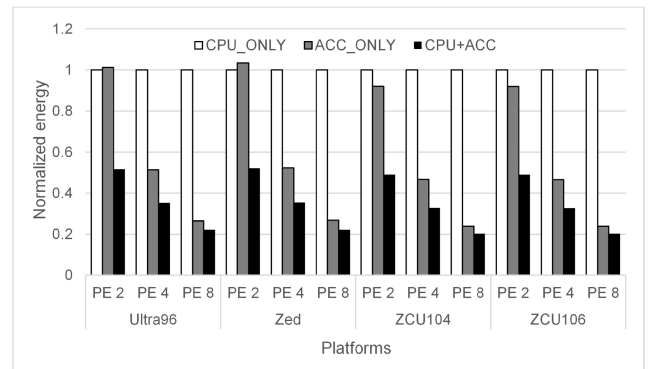


FIGURE 11. Energy results of ACC_ONLY and CPU+ACC normalized to CPU_ONLY.

output channel distribution of our co-scheduling technique based on the proposed latency model makes the CPU and accelerator to be fully utilized as much as possible, thus maximizing the throughput.

C. ENERGY

We have also determined energy consumption for our proposed co-scheduling technique based on the platform-level power measured by HPM-300A power meter [20]. Figure 11 summarizes the normalized energy results of CPU_ONLY, ACC_ONLY, and CPU+ACC. Due to the reduced execution time, our proposed co-scheduling approach CPU+ACC shows energy reduction across different accelerators with varying number of PEs by 48.0%–79.8% and 14.9%–49.7% as compared to CPU_ONLY and ACC_ONLY, respectively. Results indicate that the power consumption of the CPU+ACC increases as compared to CPU_ONLY and ACC_ONLY because the CPU+ACC makes the CPU and the accelerator to stay in the active state most of the time during the CONV layer operations. In fact, CPU+ACC shows higher power consumption than CPU_ONLY and ACC_ONLY by up to 1.8% and 1.3%, respectively, when we

TABLE 6. Per-layer latency results (in seconds) for CNN inferences with Tiny Darknet model [18] across CPU_ONLY, ACC_ONLY, and CPU+ACC.

CNN Layer #	Description	CPU_ONLY	ACC_ONLY	CPU+ACC
Layer 0	3×3 CONV	1.283221	1.057395	0.736854
Layer 1	2×2 max pooling	0.073256	0.073281	0.073755
Layer 2	3×3 CONV	3.039238	2.644233	1.526805
Layer 3	2×2 max pooling	0.035911	0.035959	0.036173
Layer 4	1×1 CONV	0.083625	0.078752	0.051595
Layer 5	3×3 CONV	2.895266	2.642671	1.419424
Layer 6	1×1 CONV	0.317731	0.328591	0.225072
Layer 7	3×3 CONV	2.894089	2.642836	1.419433
Layer 8	2×2 max pooling	0.035706	0.035747	0.035734
Layer 9	1×1 CONV	0.158447	0.148602	0.085157
Layer 10	3×3 CONV	2.841999	2.623150	1.401368
Layer 11	1×1 CONV	0.313930	0.294245	0.167850
Layer 12	3×3 CONV	2.841940	2.623151	1.401373
Layer 13	2×2 max pooling	0.017782	0.017908	0.017954
Layer 14	1×1 CONV	0.157549	0.147017	0.078937
Layer 15	3×3 CONV	2.830288	2.617322	1.373944
Layer 16	1×1 CONV	0.313590	0.292231	0.156487
Layer 17	3×3 CONV	2.830093	2.617305	1.373931
Layer 18	1×1 CONV	0.627162	0.583194	0.311537
Layer 19	1×1 CONV	1.239286	1.153453	0.609620
Layer 20	14×14 avg pooling	0.008565	0.008591	0.008588
Layer 21	Softmax	0.000197	0.000197	0.000197
	Total	24.838871	22.665831	12.511788

use 8PE accelerator version. However, the reduced execution time overwhelms the increased power consumption, resulting in huge energy reductions.

D. CASE STUDY: TINY DARKNET CNN INFERENCE

In this subsection, we show CNN inference latency results for all the layers in the Tiny Darknet model [18]. As in the previous subsections, we compare the results of *CPU_ONLY*, *ACC_ONLY*, and *CPU+ACC* when executing a CNN inference with the Tiny Darknet model. For a comprehensive analysis, we breakdown the latencies of each layer in Tiny Darknet across *CPU_ONLY*, *ACC_ONLY*, and *CPU+ACC* as shown in Table 6. Since our target platform is a resource-constrained edge, we use 2PE CNN accelerator in the *ACC_ONLY* and *CPU+ACC*. We use pre-trained weight in Darknet framework and ImageNet dataset [19] for our input. The results verify that our proposed *CPU+ACC* leads to a latency reduction of CNN inference (i.e., performance improvement) by 49.6% and 44.8% as compared to *CPU_ONLY* and *ACC_ONLY*. Since our acceleration technique only accelerates the CONV layer, the latency of max pooling (Layers 1, 3, 8, and 13), average pooling (Layer 20), and softmax (Layer 21) layers in *CPU+ACC* is similar to the latency in the case of the *CPU_ONLY* and *ACC_ONLY* (i.e., the layers except for the CONV layers are performed in the CPU). However, since sixteen layers in the Tiny Darknet model are CONV layers that accounts for 73% of the total layers, our proposed *CPU+ACC* results in a huge CNN inference performance improvement of $1.81\times - 1.99\times$ as compared to the *CPU_ONLY* and *ACC_ONLY*. Please note that the latency of the *ACC_ONLY* in Layer 6 is a little higher than the *CPU_ONLY*, which is not consistent with the other CONV layers (i.e., for the CONV layers except for Layer 6, the latency of the *ACC_ONLY* is lower than that of the *CPU_ONLY*). This is because we trigger the accelerator twice in Layer 6 due to the limited on-chip memory size

(i.e., BRAM size), resulting in much longer data transfer latency.

V. RELATED WORK

Recently, as CNNs have gained a huge attention due to their wide range of AI applications, it calls for hardware accelerators [21], [22] that execute CNNs with better performance and energy efficiency. Our work is complementary to the AI hardware accelerator development efforts and our work can be applied to any of the developed CNN hardware accelerators because the main characteristics of our work is to utilize both accelerator and CPU to improve performance and energy efficiency when executing the CNN workloads. Though there have been many proposals for hardware accelerators, system-level approaches that target co-scheduling of CNN workload on accelerators and CPU have not been fully investigated yet.

In [2], a framework to utilize heterogeneous hardware resources in SoCs when executing the CNN inferences is proposed. It utilizes CPU (with SIMD engine) and accelerator to expedite multiple CONV layer executions for different image frames. In [23], a technique to utilize CPU, GPU, and hardware accelerator for CNN acceleration with partitioning a batch of images is proposed. By exploiting the roofline model, the technique partitions a batch of images and distributes images to CPU, GPU, and FPGA accelerator. In [24], a task assignment technique is proposed for multi-CNN acceleration, which utilizes multiple deep learning processing units (DPUs) for CNN inference while CPU is responsible for task initialization. The previously proposed approaches targeting heterogeneous hardware utilization for CNN inferences are similar to our work in that they use multiple available resources (CPU and FPGAs in [2], CPU, GPU, and FPGAs in [23], and CPU and multiple DPUs in [24]); however, the previously proposed approaches can only be employed when simultaneously executing multiple CNN inferences, thus limiting their applicability. On the contrary, our proposed technique can be applied to the single CONV layer acceleration, which has wider applicability as compared to [2], [23], and [24]. In addition, those coarse-grained task partitioning may not work well in resource-constrained edge devices because it is very rare to execute a large batch of images together in edge devices. In contrast, our work employs a finer-grained approach that distributes output channels in a single CONV layer to the accelerator and CPU, which is more suitable for resource-constrained edge devices. Similarly [25] proposes a technique to utilize both FPGA-based accelerator and CPU. The technique [25] offloads the convolution operation to the FPGA accelerator while the other parts (such as fully connected layer or shortcut connection, etc.) are executed in the CPU. Though that technique can improve throughput of the CNN inference by exploiting both CPU and FPGA-based accelerator, it is very hard to fully utilize both hardware accelerator and CPU because granularity for task distribution is still large (e.g., layer granularity). Further, due to coarse granularity,

there is a high probability for having idle period in hardware resources, thus incurring a throughput loss. In contrast, our work is based on a task distribution with a finer granularity (i.e., CNN output channels in a CONV layer), and thus has a higher probability of keeping the accelerator and the CPU busy as compared to prior works. In addition, our accurate latency estimation guides the load-balanced task distribution to the accelerator and CPU, thereby minimizing idle time in the hardware resources when executing the CONV layer operations. In [26], a single layer acceleration technique is proposed by utilizing both CPU and GPU. However, our target domain for this work is resource-constrained edge devices, which do not have GPUs. We note that some edge devices do have higher computing resources and can assimilate GPUs where the technique proposed in [26] is applicable; however, our current work focuses on CPU and the hardware accelerator co-scheduling. Moreover, the technique in [26] distributes the output channels with a fixed ratio (e.g., 0.25, 0.5, and 0.75), which may lead to load imbalance between multiple hardware resources. On the contrary, our proposed technique attains nearly complete load balancing based on our accurate latency model. In [16], a technique to accelerate memory streaming workloads (memory-to-memory data transfer with or without simple arithmetic operations such as memory copy, add, scale, and triad) via a cooperation between CPU and FPGA-based accelerator is proposed. Though that technique can accelerate 1×1 convolution operations, it has limitations on accelerating $N \times N$ convolution where $N > 1$. On the contrary, our work can be applied to any type of convolution operations.

VI. CONCLUSIONS

Recent trend of artificial intelligence (AI) at the edge has resulted in assimilation of AI hardware accelerators in edge devices and edge servers for efficient AI inference. Convolutional neural network (CNN) acceleration at the edge has, in particular, gained tremendous attention as CNN acceleration at the edge can help enable many novel applications such as autonomous vehicles, surveillance, and robots. In typical resource-constrained edge systems, while hardware accelerators are running, CPUs remain idle. However, the CPU can also contribute to the CONV layer execution along with the accelerator, which can further improve performance and energy efficiency. In this paper, we have proposed a CPU-accelerator co-scheduling technique to accelerate a single CONV layer operation during the CNN inference at the edge. By exploiting the independence among the operations for generating different CNN output channels, our co-scheduling technique distributes the output channels to the accelerator and CPU, which leads to further performance improvements as compared to the accelerator-only execution. For load balancing between the accelerator and CPU, we have also proposed a linear regression-based latency model which can estimate the CONV layer execution time on the CPU and the accelerator. Based on the latency estimation from our model, we can

distribute the output channels in a load-balanced manner so that the accelerator and the CPU can be fully utilized. Our proposed technique helps in minimizing the idle time in the accelerator and the CPU, resulting in performance improvements. We have implemented our technique in four different FPGA-SoC platforms as a proof-of-concept. Experimental results indicate that our proposed co-scheduling technique improves system performance by $1.18 \times - 2.00 \times$ compared to the accelerator-only execution across a wide spectrum of the accelerator platforms. Moreover, our technique also reduces platform-level energy consumption by $14.9\% - 49.7\%$ as compared to the accelerator-only execution. In addition, as demonstrated in our case study, a full layer CNN inference for Tiny Darknet CNN model with our co-scheduling technique shows $1.81 \times - 1.99 \times$ better performance as compared to that without our technique. In future, we plan to extend this work to include co-scheduling of multiple heterogeneous resources (i.e., CPUs, GPUs, accelerators, etc.) for CNN acceleration.

REFERENCES

- [1] B. Gu, J. Kong, A. Munir, and Y. G. Kim, "A framework for distributed deep neural network training with heterogeneous computing platforms," in *Proc. IEEE 25th Int. Conf. Parallel Distrib. Syst. (ICPADS)*, Dec. 2019, pp. 430–437.
- [2] G. Zhong, A. Dubey, C. Tan, and T. Mitra, "Synergy: An HW/SW framework for high throughput CNNs on embedded heterogeneous SoC," *ACM Trans. Embedded Comput. Syst.*, vol. 18, no. 2, pp. 1–23, Apr. 2019.
- [3] S. Han, J. Pool, J. Tran, and W. J. Dally, "Learning both weights and connections for efficient neural network," in *Proc. Adv. Neural Inf. Process. Syst., Annu. Conf. Neural Inf. Process. Syst.*, 2015, pp. 1135–1143.
- [4] S. Han, H. Mao, and W. J. Dally, "Deep compression: Compressing deep neural network with pruning, trained quantization and Huffman coding," in *Proc. 4th Int. Conf. Learn. Represent., ICLR*, 2016, pp. 1–14.
- [5] F. N. Iandola, S. Han, M. W. Moskewicz, K. Ashraf, W. J. Dally, and K. Keutzer, "SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and <0.5 MB model size," 2016, *arXiv:1602.07360*. [Online]. Available: <http://arxiv.org/abs/1602.07360>
- [6] *DarkNet: Open Source Neural Networks in C*. Accessed: Jul. 1, 2019. [Online]. Available: <https://pjreddie.com/darknet/>
- [7] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going deeper with convolutions," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit. (CVPR)*, Jun. 2015, pp. 1–9.
- [8] *Google Coral Dev Board*. Accessed: Aug. 21, 2020. [Online]. Available: <https://coral.ai/products/dev-board/>
- [9] K. Guo, L. Sui, J. Qiu, J. Yu, J. Wang, S. Yao, S. Han, Y. Wang, and H. Yang, "Angel-eye: A complete design flow for mapping CNN onto embedded FPGA," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 37, no. 1, pp. 35–47, Jan. 2018.
- [10] Y. Li, S. Ma, Y. Guo, R. Xu, and G. Chen, "Configurable CNN accelerator based on tiling dataflow," in *Proc. IEEE 9th Int. Conf. Softw. Eng. Service Sci. (ICSESS)*, Nov. 2018, pp. 309–313.
- [11] Y. Zhao, X. Chen, Y. Wang, C. Li, H. You, Y. Fu, Y. Xie, Z. Wang, and Y. Lin, "SmartExchange: Trading higher-cost memory storage/access for lower-cost computation," in *Proc. ACM/IEEE 47th Annu. Int. Symp. Comput. Archit. (ISCA)*, May 2020, pp. 1–14.
- [12] *Linaro Ultra96 Evaluation Board*. Accessed: Jun. 16, 2020. [Online]. Available: <https://www.96boards.org/product/ultra96/>
- [13] *Avnet*. Accessed: Apr. 7, 2020. [Online]. Available: <http://www.zedboard.org/>
- [14] *Xilinx Zcu104 Evaluation Board*. Accessed: Jul. 20, 2020. [Online]. Available: <https://www.xilinx.com/products/boards-and-kits/zcu104.html>
- [15] *Xilinx Zcu106 Evaluation Board*. Accessed: Jul. 20, 2020. [Online]. Available: <https://www.xilinx.com/products/boards-and-kits/zcu106.html>

- [16] K. Lee, J. Kong, Y. G. Kim, and S. W. Chung, "Memory streaming acceleration for embedded systems with CPU-accelerator cooperative data processing," *Microprocessors Microsyst.*, vol. 71, Nov. 2019, Art. no. 102897.
- [17] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov, and L.-C. Chen, "MobileNetV2: Inverted residuals and linear bottlenecks," in *Proc. IEEE/CVF Conf. Comput. Vis. Pattern Recognit.*, Jun. 2018, pp. 4510–4520.
- [18] *Tiny Darknet*. Accessed: Oct. 11, 2020. [Online]. Available: <https://pjreddie.com/darknet/tiny-darknet/>
- [19] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "ImageNet: A large-scale hierarchical image database," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, Jun. 2009, pp. 248–255.
- [20] *Adpower. Hpm-300A Digital Power Meter and Analyzer*. Accessed: Mar. 13, 2020. [Online]. Available: <http://adpower21.com/>
- [21] K. Lee, J. Kong, and A. Munir, "Hw/sw co-design of cost-efficient CNN inference for cognitive IoT," in *Proc. IEEE Int. Conf. Intell. Comput. Data Sci. (ICDS)*, 2020, pp. 1063–1076.
- [22] V. Sze, Y.-H. Chen, T.-J. Yang, and J. S. Emer, "Efficient processing of deep neural networks: A tutorial and survey," *Proc. IEEE*, vol. 105, no. 12, pp. 2295–2329, Dec. 2017.
- [23] V. K. A. George, S. Gunisetty, S. Subramanian, S. K. R., and M. Purnaprajna, "CoIn: Accelerated CNN co-inference through data partitioning on heterogeneous devices," in *Proc. 6th Int. Conf. Adv. Comput. Commun. Syst. (ICACCS)*, Mar. 2020, pp. 90–95.
- [24] J. Zhu, L. Wang, H. Liu, S. Tian, Q. Deng, and J. Li, "An efficient task assignment framework to accelerate DPU-based convolutional neural network inference on FPGAs," *IEEE Access*, vol. 8, pp. 83224–83237, 2020.
- [25] 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," 2017, *arXiv:1712.00994*. [Online]. Available: <http://arxiv.org/abs/1712.00994>
- [26] Y. Kim, J. Kim, D. Chae, D. Kim, and J. Kim, " μ layer: Low latency on-device inference using cooperative single-layer acceleration and processor-friendly quantization," in *Proc. EuroSys*, 2019, pp. 1–15.



JOONHO KONG (Member, IEEE) received the B.S. degree in computer science from Korea University in 2007, and the M.S. and Ph.D. degrees in computer science and engineering from Korea University in 2009 and 2011, respectively. He was a Postdoctoral Research Associate with the Department of Electrical and Computer Engineering, Rice University, from 2012 to 2014. He was a Senior Engineer with Samsung Electronics from 2014 to 2015. He is currently an Associate Professor with the School of Electronics Engineering, Kyungpook National University. His research interests include computer architecture, heterogeneous computing, embedded systems, and hardware/software co-design.



ARSLAN MUNIR (Senior Member, IEEE) received the M.A.Sc. degree in electrical and computer engineering (ECE) from The University of British Columbia, Vancouver, Canada, in 2007, and the Ph.D. degree in ECE from the University of Florida, Gainesville, FL, USA, in 2012. From 2007 to 2008, he was a Software Development Engineer with the Embedded Systems Division, Mentor Graphics Corporation. He was a Postdoctoral Research Associate with the Department of ECE, Rice University, Houston, TX, USA, from 2012 to 2014. He is currently an Assistant Professor with the Department of Computer Science, Kansas State University.

His current research interests include embedded and cyber-physical systems, secure and trustworthy systems, parallel computing, reconfigurable computing, and artificial intelligence safety and security. He received many academic awards, including the Ph.D. Fellowship from the Natural Sciences and Engineering Research Council of Canada. He received gold medals for best performance in electrical engineering, and gold medals and academic roll of honor for securing rank one in pre-engineering provincial examinations (out of approximately 300 000 candidates).

...



YEONGMIN KIM (Student Member, IEEE) received the B.S. degree in electronics engineering from Kyungpook National University in 2019, where he is currently pursuing the M.S. degree in electronics engineering. His research interests include convolutional neural network acceleration, mobile system-on-chip design, and field-programmable gate array-based design.