2023 IEEE 29th International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA) | 979-8-3503-3786-0/23/$31.00 ©2023 IEEE | DOI: 10.1109/RTCSA58653.2023.00011

PELSI: Power-Efficient Layer-Switched Inference

Ehsan Aghapour, Dolly Sapra, Andy D. Pimentel, and Anuj Pathania
University of Amsterdam
e.aghapour@uva.nl, d.sapra@uva.nl, a.d.pimentel@uva.nl, a.pathania@uva.nl

***Abstract*—Convolutional Neural Networks (CNNs) are now quintessential kernels within embedded computer vision applications deployed in edge devices. Heterogeneous Multi-Processor System-on-Chips (HMPSoCs) with Dynamic Voltage and Frequency Scaling (DVFS) capable components (CPUs and GPUs) allow for low-latency, low-power CNN inference on resource-constrained edge devices when employed efficiently.**

CNNs comprise several heterogeneous layer types that execute with different degrees of power efficiency on different HMPSoC components at different frequencies. We propose the first framework, *PELSI*, that exploits this layer-wise power efficiency heterogeneity for power-efficient CPU-GPU layer-switched CNN interference on HMPSoCs. *PELSI* executes each layer of a CNN on an HMPSoC component (CPU or GPU) clocked at just the right frequency for every layer such that the CNN meets its inference latency target with minimal power consumption while still accounting for the power-performance overhead of multiple switching between CPU and GPU mid-inference. *PELSI* incorporates a Genetic Algorithm (GA) to identify the near-optimal CPU-GPU layer-switched CNN inference configuration from within the large exponential design space that meets the given latency requirement most power efficiently.

We evaluate *PELSI* on *Rock-Pi* embedded platform. The platform contains an *RK3399Pro HMPSoC* with DVFS-capable CPU clusters and GPU. Empirical evaluations with five different CNNs show a 44.48% improvement in power efficiency for CNN inference under *PELSI* over the state-of-the-art.

***Index Terms*—Low-Power Design, Edge Computing, Embedded Machine Learning (ML), On-Chip Artificial Intelligence (AI).**

I. Introduction

Computer vision tasks are now integral in many high-performance embedded applications across domains such as autonomous driving [10], intelligent robotics [6], image classification [9], and object detection [11]. Convolutional Neural Network (CNN) kernels processing (inference) image streams to extract recognizable features accurately are used extensively for computer vision tasks in embedded platforms. CNNs are increasingly inferencing higher resolution image streams streaming at ever-increasing frame rates. Nevertheless, privacy and performance constraints mandate the inference on the embedded platforms themselves. Embedded platforms, however, are more severely constrained in terms of their power consumption than their non-embedded counterparts. Therefore, the success of embedded computer vision applications hinges on the platforms to provide low-power, low-latency CNN inference.

Heterogeneous Multi-Processor Systems on Chips (HMPSoCs) nowadays power most high-end embedded platforms. Figure 1 illustrates the modern *RK3399Pro* HMPSoC within the *Rock-Pi N10* embedded platform. It comprises two multi-core CPU clusters – *Little* and *big*, and an embedded multi-core GPU. Dynamic Voltage and Frequency Scaling (DVFS) technology allows the HMPSoC components (CPU clusters and GPU) to run independently at different frequencies [15], [16]. Furthermore, all the HMPSoC components are capable of performing CNN inference [21]. Therefore, DVFS allows a trade-off between

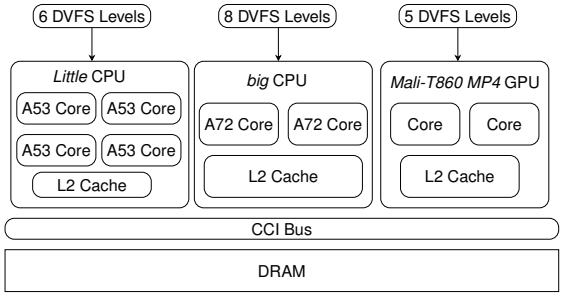
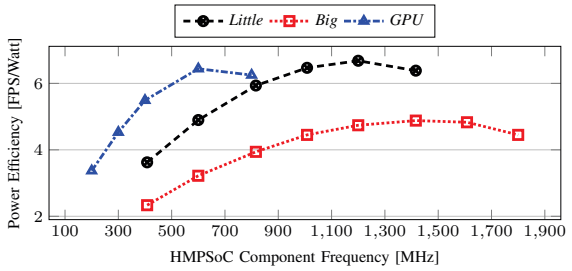
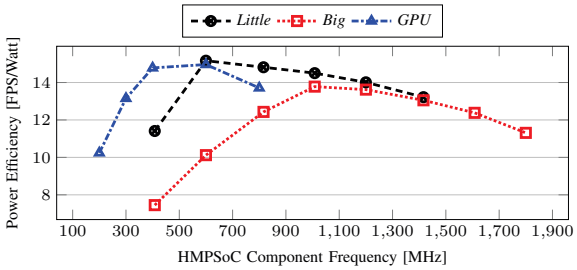


Fig. 1: An abstract block diagram of *RK3399Pro* HMPSoC in *RockPi N10* embedded platform.



(a) Compute-Intensive Convolution Layer



(b) Memory-Intensive Fully-Connected Layer

Fig. 2: Power efficiency of different CNN layers (from *AlexNet*) in different HMPSoC components at different frequencies.

inference performance and inference power on all HMPSoC components. The power efficiency of a CNN layer depends upon the interaction between the memory-compute characteristics of the layer and the underlying HMPSoC component. Consequently, there is a wide power-efficiency spectrum wherein one can perform a CNN inference on HMPSoCs, as shown in Figure 2. Figures 2(a) and 2(b) show changes in power efficiency with frequency change on different HMPSoC components for a compute-intensive convolution layer and memory-intensive fully-connected layer, respectively.

2325-1301/23/$31.00 ©2023 IEEE
DOI 10.1109/RTCSA58653.2023.00011

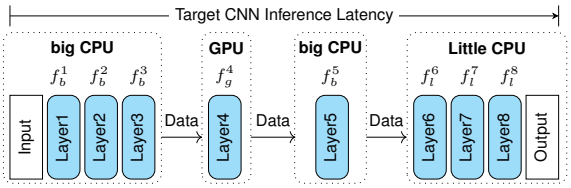


Fig. 3: An abstraction depicting power-efficient CPU-GPU layer-switched CNN inference employed within *PELSI* framework.

TABLE I: The size of design space for power-efficient CPU-GPU layer-switched inference for different CNNs with different numbers of partitionable layers on *RK3399Pro* HMPSoC.

CNN	*Partitionable Layers*	*Number of Design Points*
AlexNet	8	1.7e+10
GoogleNet	11	1.2e+14
MobileNet	14	8.0e+17
ResNet50	18	1.0e+23
SqueezeNet	10	6.1e+12

The power-efficiency spectrum widens further with the possibility of performing CPU-GPU layer-switched inference on HMPSoCs. We [1] were the first to show that heterogeneous layers within a CNN exhibit performance heterogeneity on embedded CPUs and GPUs. We show some layers execute faster on the embedded CPU while others execute faster on the embedded GPU. Based on this observation, we proposed a framework to switch between CPU and GPU mid-inference based on the executing layer for maximizing the performance of CNN inference. We showed that even after accounting for the overhead of switching between CPU and GPU back-and-forth, the observed latency of CNN inference was lower than executing purely on either of the HMPSoC components. However, we do not explore DVFS with layer-switched inference and run their CPU and GPU only at their maximum frequencies for the highest inference performance. But execution only at the highest frequency results in a high-power inference. Combining CPU-GPU DVFS with CPU-GPU layer-switched inference allows for a fine-grained trade-off between the CNN inference performance and power consumption under a given latency constraint. Note, the layer-switched inference design is distinct from pipelined CNN inference design [20].

We introduce a novel framework, *PELSI*, that explores the idea of power-efficient CPU-GPU layer-switched inference, as shown in Figure 3. CNN inference under *PELSI* switches between HMPSoC components mid-inference depending upon the CNN layer under execution. *PELSI* simultaneously sets the layer-wise frequency of the HMPSoC component to the level that allows the CNN to achieve its target latency most power efficiently. *PELSI* operates within a large exponential design space. HMPSoC with F_B, F_L, and F_G DVFS frequency levels for its *big* CPU, *Little* CPU, and GPU projects a design space of size $(F_B + F_L + F_G)^N$ for a CNN with N partitionable layers. Table I shows the design space size for different CNNs on a *RK3399Pro* HMPSoC. It is computationally infeasible to exhaustively search this design space for an optimal solution. Therefore, *PELSI* includes a Genetic Algorithm (GA) to identify a near-optimal solution for power-efficient CPU-GPU layer-switched inference under latency constraints.

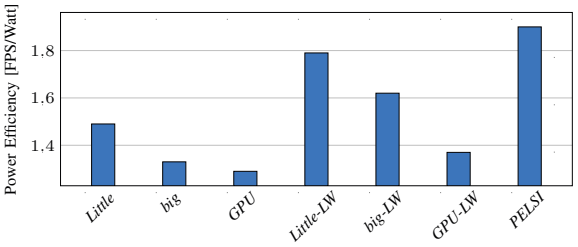


Fig. 4: Power efficiency of *MobileNet* running on *Little CPU*, *big* CPU, and GPU with fixed DVFS (*Little*, *big*, and *GPU*), layer-wise DVFS (*Little-LW*, *big-LW*, and *GPU-LW*), and the proposed *PELSI* framework under a given latency constraint of 200 ms

Motivational Example: We perform a CNN inference with *MobileNet* with a target latency of 200 ms as a motivational example. Figure 4 shows the power efficiency of CNN inference that meets the target latency with different executions. The CNN inference attains a power efficiency of 1.49 FPS/Watt, 1.33 FPS/Watt, and 1.29 FPS/Watt while executing at the lowest possible fixed frequency that meets the target latency using only *Little* CPU, *Big* CPU, and *GPU*, respectively.

We now allow layer-wise DVFS for single-component CNN inference and use a configuration that meets the target latency. However, we run different CNN layers at different frequencies to maximize the inference's power efficiency on a single HMPSoC component. Figure 4 shows that the CNN inference attains a power efficiency of 1.79 FPS/Watt, 1.62 FPS/Watt, and 1.37 FPS/Watt when executing layer-wise DVFS that meets the target latency while using *Little* CPU, *big* CPU, and *GPU*, respectively. Using layer-wise DVFS increases the power efficiency of inference by 20.00%, 21.89%, and 6.27% against fixed-frequency inference on *Little* CPU, *big* CPU, and *GPU*, respectively.

Finally, in our motivational example, we allow layer-wise DVFS alongside switching between HMPSoC components mid-inference as proposed with *PELSI*. Figure 4 shows that CNN inference under *PELSI* attains a power efficiency of 1.89 FPS/Watt for the same target latency. *PELSI* increases the power efficiency of CNN inference by 6.26%, 17.43%, and 38.42% against single-component layer-wise DVFS CNN inference on *Little* CPU, *big* CPU, and *GPU*, respectively. Therefore, our motivational example motivates using layer-wise DVFS synergistically with HMPSoC component-switching for extracting maximum power efficiency for a latency-constrained CNN inference.

Our Novel Contributions: We make the following novel contributions in the context of this work.

- We propose a framework, *PELSI*, that explores the idea of power-efficient CPU-GPU layer-switched CNN inference.
- We propose a GA within *PELSI* to identify a near-optimal configuration for power-efficient CPU-GPU layer-switched CNN inference under latency constraints.
- We implement the proposed *PELSI* framework within the *ARM Compute Library* (*ARM-CL*) and evaluate it using *RK3399Pro HMPSoC*.

Open Source Contribution: *PELSI* is publicly available for download at *https://github.com/Ehsan-aghapour/ARMCL-pipe-all ("CPU-GPU-LW" branch).*

TABLE II: The available DVFS levels of the *Little* CPU, *big* CPU, and GPU on *RK3399Pro* HMPSoC.

***Little* CPU**		***big* CPU**		**GPU**	
Frequency (MHz)	Voltage (mV)	Frequency (MHz)	Voltage (mV)	Frequency (MHz)	Voltage (mV)
408	800	408	800	200	800
600	800	600	800	300	800
816	850	816	825	400	825
1008	925	1008	875	600	925
1200	1000	1200	950	800	1100
1416	1125	1417	1025		
		1608	1100		
		1800	1200		

II. Related Work

Efficient power management has always been an important consideration for resource-constrained embedded devices. In this respect, various hardware-based techniques have been proposed in the literature, from power monitoring [5] and power modeling [3] for off-the-shelf hardware to creating special low-powered hardware [2]. However, only hardware-level techniques can not effectively minimize power consumption unless accompanying software-level techniques exploit application domain-specific knowledge for power efficiency [19]. We formulate *PELSI* with CNN-specific knowledge considerations and its per-layer behavior on different computing resource types.

In research, several works improve power efficiency for CNN training [4], [8]. However, a trained CNN deployed on an embedded device for a long duration will benefit more from the power efficiency of the inference. Another popular direction to achieve power efficiency during CNN inference is to search for appropriate CNN models through Neural Architecture Search (NAS) algorithms [12], [13]. In principle, *PELSI* is orthogonal to such techniques. One can use *PELSI* for an efficient layer-switched inference for a CNN designed through NAS methodologies.

Traditionally, in High-Performance Computing (HPC), CNN inference is performed solely on GPUs. Therefore, multiple works only focus on power efficiency for inference on GPUs, such as [18] and [22]. *PELSI*, on the other hand, focuses on HMPSoCs where embedded CPUs and GPUs are comparable in performance, and both are used for inference to maximize efficiency. To this end, a few works [7] propose to analyze the power efficiency of CNN inference on HMPSoCs for both CPUs and GPUs. The authors of [3] propose a per-core power model, while the authors of [17] analyze power-performance profiles of various CNNS for CPU and GPU platforms. However, unlike *PELSI*, these papers do not propose any tangible solution to improve power efficiency through hardware-software co-design.

DVFS is well-known as an efficient methodology for power management on embedded devices. For example, [18] reduces the energy consumption of CNN training on a GPU by controlling its operational frequency. *AOA* [14] coordinates the frequency of both CPU and GPU to improve performance and reduce total energy consumption. *AOA* achieves this by balancing the workload at a higher abstraction level and does not delve deeper into the working of a CNN and its layers.

III. Setup

We use *Rock-Pi N10* embedded platform in this work. An *RK3399Pro HMPSoC*, as shown in Figure 1, powers the *Rock-Pi N10 platform*. The platform has a hexa-core asymmetric

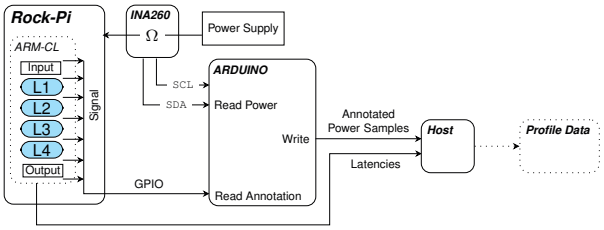


Fig. 5: An abstract diagram depicting setup for CNN inference layer-level power profiling.

ARM big.Little multi-core CPU with two CPU clusters – *big* and *Little*. The *Little* CPU cluster contains four low-power, low-performance *A53* cores with six DVFS levels. The *big* CPU cluster contains two high-performance, high-power *A72* cores with eight DVFS levels. It also has an *ARM* dual-core *Mali T860* GPU with six DVFS levels. Table II summarizes all the DVFS levels available for different components in the *RK3399Pro HMPSoC*. A 4 GB LPDDR4 acts as the main memory https://www.overleaf.com/project/64024988b5f20ae00aed5297for the HMPSoC. The platform is running *Android* v9.0 with kernel v4.9.

We use *ARM-CL* v21.02 in this work for CNN inference. We use *AlexNet*, *GoogleNet*, *MobileNet*, *ResNet50*, and *SqueezeNet* as the CNN kernels. These CNNs perform image classification. Their designers trained them on the *ImageNet* data set for 1000 image classes. The input to these models are images of size (224 × 224) with three channels (RGB), and the output is a tensor of size 1000 predicting the input image's class. We implement the GA using *NSGA2* employing the *pymoo Python3* library.

We use an external power data acquisition setup for fine-grained CNN inference power consumption measurements on *Rock-Pi N10*, as shown in Figure 5. The setup uses an *INA260* sensor that measures the current and voltage transfer over a single `I2C` interface. We pass the power supply for the *Rock-Pi N10* through an *INA260* sensor, wherein the sensor measures the current based on dropped voltage with an internal shunt resistor. An *Arduino Uno* embedded board samples voltage and current readings from the *INA260* sensor through an `I2C` clock and data pins (`SCL` and `SDA`) at the sampling rate of 692 samples per second. *Arduino Uno* sends the data to the host laptop for further processing over the serial (USB) port. The setup allows the annotation of the power data with meta-data using signals passed from the *Rock-Pi N10* through GPIO pins to the *Arduino Uno*. We modify *ARM-CL* to send a signal (with meta-data) at the start and end of the execution of each layer of the CNN. The meta-data then allow us to separate the power consumption of each layer of CNN during an inference. We get the corresponding layer latency directly from *ARM-CL* using clock functions.

IV. Implementation

We implement the proposed *PELSI* framework within the *ARM-CL* framework. *ARM-CL* is a collection of low-level (written in *Assembly* language) Machine Learning (ML) functions. They come highly optimized for the *ARM Cortex-A* CPU and *Mali* GPU cores. *ARM-CL* forms the ideal choice to perform CNN inference for our setup. Power-efficient CPU-GPU layer-switched CNN inference involves implementing two new features – HMPSoC

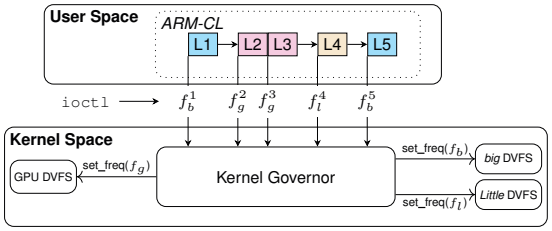


Fig. 6: An abstraction depicting the implementation of *PELSI*.

TABLE III: The min and max DVFS delay(µs) for the *Little* CPU, *big* CPU, and GPU on *RK3399Pro* HMPSoC, when transitioning to higher (up) and lower (down) frequency levels.

Transition	*PE*	*Min Delay*	*Frequency*		*Max Delay*	*Frequency*	
			i	*i+1*		*i*	*i+1*
Up	L	296	0	1	4211	0	2
	B	193	0	1	3811	6	7
	G	657	0	1	4461	2	4
Down	L	109	4	3	193	3	0
	B	91	7	3	1413	4	1
	G	670	4	1	1464	4	2

component-switching and layer-wise DVFS – not available by default in the *ARM-CL*.

Authors of [1] extensively extend *ARM-CL* to support HMPSoC component switching and make those extensions publicly available. We start with this extended open-source version of *ARM-CL* that allows for HMPSoC component-switching out-of-the-box in this work. We add more extensions to this *ARM-CL* version to support orthogonal layer-wise DVFS enabling the proposed *PELSI*. Figure 6 shows the implementation of layer-wise DVFS working synergistically with component-switching within *PELSI*.

It is common practice to use pseudo file system *sysfs* provided by the *Linux* kernel to change the frequency of HMPSoC components (CPU or GPU cores) from the Operating Systems (OS) user space. However, there is higher overhead when changing an HMPSoC component's frequency using *sysfs* than from within the OS kernel space. Therefore, *PELSI* updates the frequencies for layer-wise DVFS change from within the kernel space to make layer-wise DVFS time-wise feasible.

The CNN inference happens in the user space. Therefore, the information when a CNN layer starts/finishes execution is only available in the user space. This information must get passed down to the kernel space to perform synchronized layer-wise DVFS with minimal overhead. We embed `ioctl` calls (written in *C/C++*) at the start of the execution of every CNN layer in *ARM-CL* that signal the execution frequency for the layer's preferred HMPSoC component. A custom power Governor of our design (embedded within the kernel source code) receives this signal in the kernel space. It then sets the frequency of the preferred HMPSoC component to the value within the received signal. Simultaneously, to minimize power consumption, it sets the frequency of the non-preferred (idle) HMPSoC components to the minimum value. The involved `ioctl` calls are non-blocking. The low overhead allows for fast layer-wise DVFS.

During experiments, we observe a noticeable delay from initiating a frequency update to the actual change in the hardware. This delay varies for different frequency levels for all components. Table III shows the minimum and maximum frequency transition delay (µs) when increasing (up transition) and decreasing frequency (down transition).*PELSI* accounts for these delays in computing the execution time and power of a layer.

Chromosome

(c_1, f_1)	(c_2, f_2)	...	(c_N, f_N)

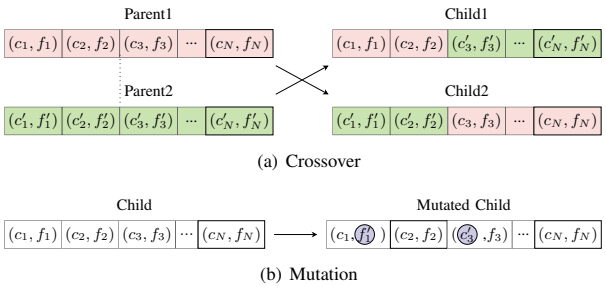
Fig. 7: Chromosome gene encoding representing the HMPSoC component type (c_i) and the corresponding component frequency (f_i) for every layer i in CNN with N layers.

V. Algorithm

PELSI, as shown in Figure 3, requires the determination of the preferred HMPSoC component and the component's corresponding layer-wise frequency for every CNN layer. *PELSI* must ensure meeting the inference's latency target with maximum power efficiency. *PELSI* targets an NP-hard optimization problem with a large exponential design space, as shown in Table I. Consequently, it is impossible to brute-force the optimal power-efficient CPU-GPU layer-switched CNN inference configuration in *PELSI*. Therefore, we propose to use a Genetic Algorithm (GA) within *PELSI* to find a near-optimal configuration that meets a given CNN inference target latency with minimal power consumption. The GA accounts for the power-performance overhead of switching components mid-inference inherent in *PELSI*. The implementation overhead for achieving fine-grained layer-level DVFS in *PELSI* (Figure 6) is negligible. Therefore, the GA does not take it into account for optimization.

It takes up to a minute to determine the power-performance attributes for a configuration in *PELSI* directly from the embedded platform. The GA within *PELSI* requires an evaluation of hundreds of thousands of such configurations. Consequently, it is time-wise infeasible (though technically possible using our setup) to run the GA directly with live power-performance feedback from the embedded platform. Therefore, we instead execute our GA using power-performance profiled data for every CNN layer at different HMPSoC components at different frequencies obtained using the data acquisition setup shown in Figure 5. We use a linear regression model that correlates the size of data migration (between the components on switching) with the observed power and performance penalty to determine the power-performance overhead of component switching involved in a configuration. This evaluation design allows the GA to find a near-optimal configuration in a reasonable amount of time at the cost of introducing a minimal error between the expected and observed power-performance attributes of the solution configuration when ported to the real embedded platform.

A Genetic Algorithm (GA) is a meta-heuristic design-space exploration algorithm based on the process of natural evolution. The proposed GA is an iterative algorithm that begins with encoding some configurations into chromosomes. Each distinct chromosome represents a unique individual. All the individuals together form the initial population. After evaluating the population, using a fitness function based on CNN inference latency and power consumption, the weak (and unviable) individuals are replaced with the offspring of the stronger individuals produced through mating (crossover and mutation) functions. This process continues over multiple iterations. Therefore, the population in later iterations will consist of fitter individuals representing more



(a) Crossover

(b) Mutation

Fig. 8: An abstraction illustrating the mating process using crossover and mutation operation within the proposed GA for a CNN with N layers.

power-efficient latency-meeting configurations than the configurations that formed the initial population. The process eventually converges to the solution population when the GA can not produce stronger offspring anymore.

Population: The population contains individuals with a single chromosome representation of configurations. Figure 7 shows the gene encoding for a chromosome within an individual. There is a gene for every layer in the CNN under inference. The gene contains the information about the HMPSoC component and the corresponding frequency with which the layer will execute on the HMPSoC. The chromosome, therefore, contains all the information necessary to determine the power-performance attribute of its underlying configuration.

Fitness function: A fitness function evaluates and ranks the offspring based on an optimization objective while operating under CNN inference latency constraint. The optimization objective for the GA is to minimize the energy consumption per inference. Measuring the onboard energy consumption for each design point takes a long time. Therefore, to converge within a reasonable time frame, we use profiles of the time and power consumption for layers of CNN when running on Little CPU, big CPU, and GPU at all frequency levels. The fitness function estimates the layer-wise energy consumption and inference latency using a regression model on the profiled data. Moreover, there is a transition delay by the request to switch frequency while the layer has already started to execute. This delay leads to part of the layer executing in the old frequency until the new frequency is active in the hardware.

The energy estimation model accounts for various factors of the DVFS-based layer-switched inference of a CNN. The final estimation estimates power consumption and timing analysis of per-layer execution, the transition delay, and the communication overhead to switch between different processing components. We investigate the correctness of the energy consumption estimation function by measuring the actual values from 1000 random design points and comparing them against estimated values. The overall mean error rate between estimated and measured values remained between 6.16% and 9.32% for different CNN models.

Selection: After evaluating and ranking the population, the GA selects n parents ($\frac{n}{2}$ pairs) using a binary tournament selection algorithm to generate offspring with mating operations. In binary tournament selection, two individuals are randomly selected from a population and compared with each other. The GA then selects the individual with higher fitness as one of the parents for the next generation. It repeats the tournament (with the previously selected parent excluded) to find the other parent. The GA then marks the two selected parents as a mating pair. An individual can be in multiple mating pairs. Therefore, a stronger individual within the population has a higher probability of spawning more offspring for the next generation. The selection process repeats till the GA selects the desired mating pairs.

Mating: The GA uses crossover and mutation operations to generate offspring from the selected parents, as shown in Figure 8. For the crossover operation, the GA selects a random crossover point in the chromosome and generates two offspring by combining the first and second sections of each pair, as shown in Figure 8(a). Therefore, for the offspring produced from the crossover, one parent determines the HMPSoC component mapping and the corresponding DVFS settings for the first section. Complementarily, the other parent determines the mapping and DVFS setting of the second section. After generating offspring, the GA applies mutation by selecting a random gene (layer) and changing its first value (target HMPSoC component), and selecting another random gene and changing its second value (DVFS settings), as shown in Figure 8(b).

Survival: The mating process generates an offspring population Q_i in GA iteration i. The offspring population Q_i is the same size n as the parent population P_i, as each pair of parents generates exactly two offspring. The GA then merges the offspring population Q_i with the parent population P_i to generate the merged population R_i. It then ranks the individuals in the merged population R_i according to their fitness. The GA then selects the top n individuals from R_i to form the next parent generation P_{i+1}, corresponding to the GA iteration $i+1$. This process ensures that the parent generation P_{i+1} could not be worse in terms of fitness than the parent generation P_i in the previous iteration.

Convergence: The process of selection, followed by mating, and survival repeats iteratively in the GA. In each iteration with crossover, the GA explores different areas of design space. In each iteration with mutation, the GA attempts to find better design points within an area. The GA achieves convergence when it is no longer possible to produce stronger (yet viable) offspring from a given generation of parents. The GA reports the top-ranked individual as the solution on convergence. Within this individual's chromosome resides the near-optimal power-efficient configuration that meets the given latency constraint.

VI. Results

We evaluate our proposed *PELSI* framework on *Rock-Pi N10* embedded platform containing *RK3399Pro HMPSoC*. Since the work introducing the idea of CPU-GPU layer-switched execution [1] does not employ DVFS and focuses only on maximizing performance, it is not fair to compare it against it in the context of power efficiency. The work most similar to *PELSI* is *AOA* [14]. The original *AOA* utilizes DVFS at run-time based on the power and utilization of HMPSoC components to minimize total energy consumption during a CNN inference. We modify *AoA* to improve power efficiency under a latency constraint similar to *PELSI* for a baseline comparison. We call the baseline *AoA-like*.

AOA comprises two parts. The first part works on the power consumption of the current interval and the CPU-GPU utilization. *AOA* in the first part determines if it can increase the frequency of processors while keeping the total energy consumption as the constraint. In the second part, *AOA* computes an imbalance factor representing the CPU and GPU utilization difference. When this

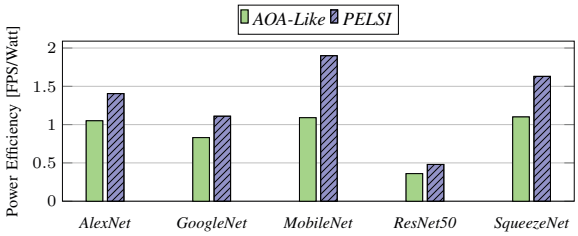


Fig. 9: Normalized CNN inference power-efficiency for *PELSI* against the state-of-the-art for different CNNs.

factor outweighs a threshold, it triggers the imbalance state. In the imbalance state, the bottleneck processor's frequency increases till it reaches the maximum level. Otherwise, *AOA* decreases the frequency of the non-bottleneck processor.

We design the *AOA-Like* algorithm by modifying the first part of the original *AOA*. *AOA* preserves total energy consumption, whereas the *AOA-Like* has to meet the latency deadline. For this purpose, after each layer execution, *AOA-Like* decides if the frequency of a component is worth increasing to meet the target latency. We measure the execution time of CNN layers with all fixed frequency combinations of GPU and its host (big CPU) to compute the average execution time per layer. With this data, AOA-Like assign every layer a *task-portion* number, which is the percentage of time taken by the layer within the overall inference time. The total *task-portion* number of remaining layers is then compared with the remaining percentage of time left in the target latency (*time-portion*). If the ratio of the total *task portion* to the *time portion* is higher than one, the frequency level is increased by the same number of steps as this ratio. The second part of AOA-Like remains unchanged to keep the CPU and GPU balance.

We evaluate the improvement in the power efficiency with *PELSI* over *AOA-Like* design with five conventional CNNs. For the target latency of each CNN, we add approximately 100 ms to the minimum latency that the *AOA-Like* design achieves. Figure 9 depicts the power efficiency of *PELSI* against the *AOA-Like* for different CNNs. It demonstrates that for all evaluated CNNs, the *PELSI* achieves higher power efficiency. These power efficiency improvements are 33.40%, 34.09%, 74.74%, 31.41%, and 48.73% for *AlexNet*, *GoogleNet*, *MobileNet*, *ResNet50*, and *SqueezeNet*, respectively. Across all CNNs, the *PELSI* framework improves power efficiency by 44.48% on average.

VII. Conclusion

We observe different CNN layers execute with different power efficiency on different HMPSoC components (CPU or GPU) running at different frequencies. Based on the observation, we propose *PELSI* framework that explores power-efficient CPU-GPU layer-switched CNN inference in this work. *PELSI* proposes to switch between CPU-GPU mid-inference while simultaneously changing the frequency of the active inferencing HMPSoC component. *PELSI* incorporates a GA to find a near-optimal power-efficient CPU-GPU layer-switched CNN inference configuration under a latency constraint. We evaluate the proposed *PELSI* framework on *Rock-Pi N10* embedded platform containing *RK3399Pro HMPSoC*. Results show significant improvement in power efficiency across different CNNs with inference under *PELSI* over the state-of-the-art.

References

[1] Ehsan Aghapour, Dolly Sapra, Andy Pimentel, and Anuj Pathania. Cpu-gpu layer-switched low latency cnn inference. In *2022 25th Euromicro Conference on Digital System Design (DSD)*, 2022.

[2] Hyochan An et al. An ultra-low-power image signal processor for hierarchical image recognition with deep neural networks. *IEEE Journal of Solid-State Circuits*, 56(4), 2020.

[3] Ganapati Bhat, Sumit K Mandal, Sai T Manchukonda, Sai V Vadlamudi, Ayushi Agarwal, Jun Wang, and Umit Y Ogras. Per-core power modeling for heterogenous socs. *Electronics*, 10(19), 2021.

[4] TaiYu Cheng et al. Minimizing power for neural network training with logarithm-approximate floating-point multiplier. In *2019 29th international symposium on power and timing modeling, optimization and simulation (PATMOS)*. IEEE, 2019.

[5] Luca Cremona, William Fornaciari, and Davide Zoni. Automatic identification and hardware implementation of a resource-constrained power model for embedded systems. *Sustainable Computing: Informatics and Systems*, 29, 2021.

[6] Jonatan S Dyrstad et al. Grasping virtual fish: A step towards robotic deep learning from demonstration in virtual reality. In *2017 IEEE International Conference on Robotics and Biomimetics (ROBIO)*, pages 1181–1187. IEEE, 2017.

[7] Xiaotian Guo, Andy D. Pimentel, and Todor Stefanov. Automated exploration and implementation of distributed cnn inference at the edge. *IEEE Internet of Things Journal*, 10(7):5843–5858, 2023.

[8] Ali HeydariGorji et al. Stannis: low-power acceleration of dnn training using computational storage devices. In *2020 57th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 2020.

[9] Andrew G Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *arXiv preprint arXiv:1704.04861*, 2017.

[10] Shih-Chieh Lin, Yunqi Zhang, Chang-Hong Hsu, Matt Skach, Md E Haque, Lingjia Tang, and Jason Mars. The architectural implications of autonomous driving: Constraints and acceleration. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 751–766, 2018.

[11] Wei Liu et al. Ssd: Single shot multibox detector. In *European conference on computer vision*, pages 21–37. Springer, 2016.

[12] Svetlana Minakova et al. Scenario based run-time switching for adaptive cnn-based applications at the edge. *ACM Transactions on Embedded Computing Systems (TECS)*, 21(2), 2022.

[13] Mohanad Odema et al. Eexnas: Early-exit neural architecture search solutions for low-power wearable devices. In *2021 IEEE/ACM International Symposium on Low Power Electronics and Design (ISLPED)*. IEEE, 2021.

[14] Zhixin Ou et al. Aoa: Adaptive overclocking algorithm on cpu-gpu heterogeneous platforms. In *Algorithms and Architectures for Parallel Processing: 22nd International Conference, ICA3PP 2022, Copenhagen, Denmark, October 10–12, 2022, Proceedings*. Springer, 2023.

[15] Anuj Pathania, Alexandru Eugen Irimiea, Alok Prakash, and Tulika Mitra. Power-performance modelling of mobile gaming workloads on heterogeneous mpsocs. In *Proceedings of the 52nd Annual Design Automation Conference*, pages 1–6, 2015.

[16] Anuj Pathania, Qing Jiao, Alok Prakash, and Tulika Mitra. Integrated cpu-gpu power management for 3d mobile games. In *Proceedings of the 51st Annual Design Automation Conference*, pages 1–6, 2014.

[17] Yuyang Sun, Zhixin Ou, Juan Chen, Xinxin Qi, Yifei Guo, Shunzhe Cai, and Xiaoming Yan. Evaluating performance, power and energy of deep neural networks on cpus and gpus. In *Theoretical Computer Science: 39th National Conference of Theoretical Computer Science, NCTCS 2021, Yinchuan, China, July 23–25, 2021, Revised Selected Papers 39*. Springer, 2021.

[18] Zhenheng Tang, Yuxin Wang, Qiang Wang, and Xiaowen Chu. The impact of gpu dvfs on the energy and performance of deep learning: An empirical study. In *Proceedings of the Tenth ACM International Conference on Future Energy Systems*, 2019.

[19] Yigit Tuncel, Sizhe An, Ganapati Bhat, Naga Raja, Hyung Gyu Lee, and Umit Ogras. Voltage-frequency domain optimization for energy-neutral wearable health devices. *Sensors*, 20(18), 2020.

[20] Siqi Wang, Gayathri Ananthanarayanan, Yifan Zeng, Neeraj Goel, Anuj Pathania, and Tulika Mitra. High-throughput cnn inference on embedded arm big. little multicore processors. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 39(10):2254–2267, 2019.

[21] Siqi Wang, Anuj Pathania, and Tulika Mitra. Neural network inference on mobile socs. *IEEE Design & Test*, 37(5):50–57, 2020.

[22] Junyeol Yu, Jongseok Kim, and Euiseong Seo. A dnn inference latency-aware gpu power management scheme. In *2021 IEEE 3rd Eurasia Conference on IOT, Communication and Engineering (ECICE)*. IEEE, 2021.