Early-Exit DNN Inference on HMPSoCs

Saeed Khalilian*, Ehsan Aghapour†, Nirvana Meratnia*, Andy Pimentel†, Anuj Pathania†
*Eindhoven University of Technology, {s.khalilian.gourtani, n.meratnia}@tue.nl
†University of Amsterdam, {e.aghapour, a.d.pimentel, a.pathania}@uva.nl

***Abstract*—Using Heterogeneous Multi-Processor System-on-Chips (HMPSoCs) for Deep Neural Network (DNN) inference has become commonplace in edge devices. However, reducing the DNN inference latency on resource-constrained edge devices remains a first-class constraint. Early-Exit (EE) DNNs offer variable, input-dependent inference latency, aiming to reduce average inference latency by lowering the latency of simple inputs at the cost of increased latency for complex inputs. However, the overheads introduced by exit branches reduce the potential performance gains with EE DNNs.**

Furthermore, existing CPU- or GPU-only implementations of EE DNN inference under-utilize the HMPSoC. To address this limitation, we propose a cooperative and parallel CPU-GPU execution approach[1] for EE DNN inference that effectively distributes computations across all HMPSoC processors, minimizing latency variations. Our approach allows EE DNNs to achieve reduced average latency on HMPSoCs relative to their static counterparts, significantly reducing average- and worst-case inference latencies and enhancing the speed-up of EE DNN compared to the best single-processor inference. On average, the worst-case inference latency decreased by 24.8% across three commonly used EE DNNs, providing latency comparable to a static model without compromising accuracy on an *RK3399PRO* HMPSoC.

***Index Terms*—Edge Computing, Dynamic Networks, Early-Exit Networks, Inference Acceleration, Embedded Systems**

I. Introduction

Executing Deep Neural Networks (DNNs) on-chip in edge devices offers several benefits, such as improved privacy, responsiveness, availability, and efficiency [1]. However, high DNN inference latency on resource-constrained edge devices remains challenging [2]. Early-Exit (EE) DNNs reduce the overall inference latency and computational cost of edge Machine Learning (ML) tasks by adjusting computational intensity based on the complexity of the input data. An EE DNN integrates intermediate exit branches into its model, allowing it to halt inference partway if specific policies, such as inference reaching a high confidence level, are met. Therefore, they utilize fewer layers (computations) rather than processing the entire model for simple inputs. Previous studies have shown that EE DNNs [3, 4] can significantly reduce average latency. For example, in self-distillation [3], adding three exit branches to *ResNet18* on the *CIFAR100* dataset showed that the EE model can achieve a 3x speed-up over static *ResNet18*.

However, EE DNNs show higher latency for complex inputs. When the input data is complex, the EE DNN model needs to process all DNN layers and evaluate the output of all exit branches, which increases worst-case latency even more than a static network. This high latency can be problematic for mission-critical applications, such as autonomous cars, where low latency is crucial. Moreover, in scenarios where most inputs to the model are complex, the exit branches can introduce significant latency overhead to the model. Current EE DNNs on edge devices under-utilize the system, leading to an avoidable high worst-case latency.

Heterogeneous Multi-Processor System-on-Chips (HMPSoCs) power most of the state-of-the-art edge devices. HMPSoCs tightly integrate a high-performance inference-capable Central Processing Unit (CPU) and Graphic Processing Unit (GPU) [5]. For example, an *ARM*-based HMPSoC integrates a *big.Little* asymmetric multi-core CPU with a *Mali* GPU. However, current implementations only support CPU- or GPU-only EE DNN inference, severely under-utilizing the HMPSoC and limiting potential worst-case latency. A cooperative CPU-GPU execution approach that parallelizes the EE DNN inference across the HMPSoCs can significantly improve the average- and worst-case EE DNN inference latency.

Motivational Example: Figure 1 motivates using the entire HMPSoC for EE DNN inference. A static *ResNet18* has four building (backbone) blocks, each comprising multiple DNN layers. Executing backbone blocks in parallel among HMPSoC processors is challenging because of dependencies in the data that require a serial processing approach. A static *ResNet18* on the GPU achieves the lowest DNN inference latency of 36.62 ms. In comparison, EE *ResNet18* adds three exit branches to the network, one after each backbone block except the last one, which increases the worst-case latency of *ResNet18* to 48.75 ms on the GPU in a serial execution. However, the exit branches can run parallel to the subsequent backbone blocks. Figure 1 shows a parallelized execution of EE *ResNet18* on all processors of an HMPSoC – *big* CPU, *Little* CPU, and the GPU. This approach takes advantage of the concurrent execution of exit branches and backbone blocks. It also exploits the placement of the branches and blocks to HMPSoC processors that execute them the fastest. Results show that even after accounting for the parallelization overheads, the parallelized execution can reduce the worst-case DNN inference latency for EE *ResNet18* to 35.30 ms, a 27.56% improvement over the best single-processor inference.

Our Contributions: We make the following novel contributions within the scope of this paper.

- We characterize the execution of several EE DNNs, including *ResNet18*, *ResNet50* [3], and *VGG16* [6], on an HMPSoC platform, demonstrating that conventional

[1]Source code available at: https://github.com/Saeed-Khalilian/EEDNN_on_HMPSoCs.git

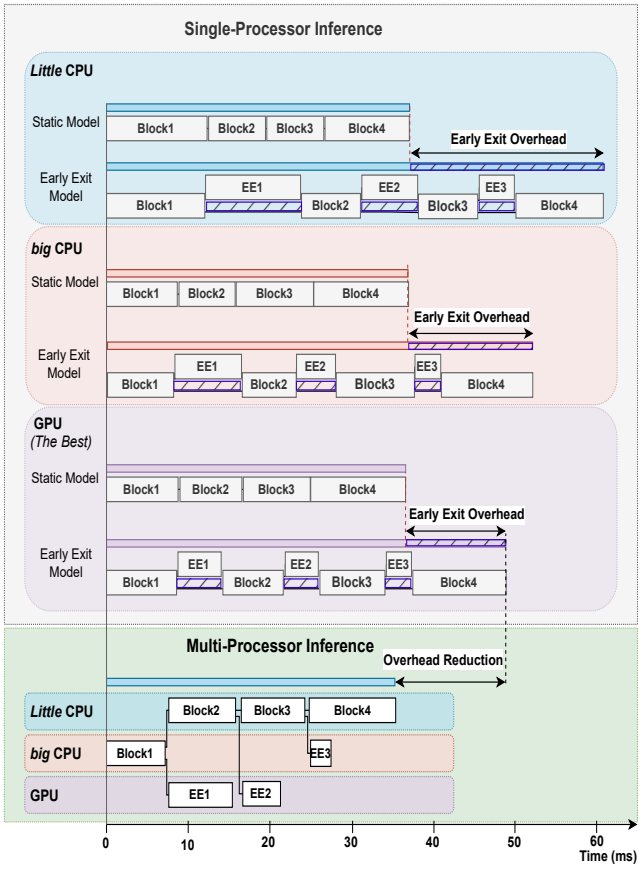

Fig. 1: Worst-case latency comparison between static *ResNet18* and EE *ResNet18* models on HMPSoCs for complex samples under serialized and parallelized executions.

deployment strategy can increase inference latency, particularly for complex samples, compared to static models.

- We analyze the performance of EE *ResNet18* with corrupted samples [7] and observe that when most samples are complex, the effectiveness of exit branches decreases significantly, leading to a noticeable latency increase.
- We propose a framework that partitions the computation graph of EE DNNs based on their latency across different HMPSoC processors and utilizes a genetic algorithm to determine the optimal processor allocation for each EE DNN component. This approach enables exit branches and backbone blocks to operate in parallel, significantly reducing both the average and worst-case latency of EE DNNs, while having no impact on the model's accuracy.
- We evaluate the framework using *Rock PI N10* embedded board with *RK3399PRO* HMPSoC and show significant real-world latency improvements for EE DNNs.

II. Related Work

Dynamic networks [8] adjust their depth [4], width [9], routing [10], and parameters [11] during inference based on input data. EE DNNs are a sub-type of dynamic networks introduced with *BranchyNet* [4]. Initial EE DNNs incorporate additional exit branches into the backbone model. The run-time terminates inferencing with the DNN mid-inference if the prediction confidence at an exit branch exceeds a specified threshold. Numerous strategies, such as gate-classification layers [12], accuracy classification [13], decision functions [14], exit predictor [6], self-distillation [3], and EE neural architecture search [15], have been proposed to improve the performance of EE DNNs by optimizing architectures, determining the optimal exit branch position, and enhancing training and inference mechanisms. These techniques have led to next-generation EE DNNs, such as *ClassyNet* [16], EExNAS [15], EDANAS [17], and *EENet* [18]. However, while these techniques enhance the accuracy and speed-up of EE DNNs, complex inputs can still introduce latency overhead, which results in inference latency higher than that of static DNNs.

Several works have tried to optimize the inference latency and throughput of static DNNs on HMPSoCs. The authors of [19] parallelized the computation of DNNs over *big* and *Little* CPU core clusters of *ARM big.Little* asymmetric multi-core processors for high inference throughput. The authors of [20] parallelized the DNN workload over an *ARM* CPU and an *Nvidia Tegra* GPU for high inference throughput. The authors of [21] parallelized the computation of DNNs over an *ARM big.Little* asymmetric multi-core processor and a *Mali* GPU for low latency. The authors of [22] parallelized the calculations of multiple DNNs running simultaneously over an *ARM big.Little* asymmetric multi-core processor and a *Mali* GPU for high throughput. The study in [23] examines 'filter sizes' in convolution kernels within DNNs as an additional factor to consider when optimizing concurrent execution for efficient partitioning between the CPU and GPU on HMPSoCs. Previous research [24] showed that by considering the idle power consumption of the HMPSoC and the active power of processors, parallel execution over a shorter period results in lower overall energy consumption compared to serial execution on a single processor over a longer duration. As a result, reducing the average execution time can also benefit energy-constrained edge devices. However, previous works only parallelized the inference computations within HMPSoC processors for static DNNs, not EE DNNs. We are the first to distribute the computations of EE DNNs within HMPSoC processors to reduce inference latency.

III. EE DNN Inference on HMPSoCs

Figure 2 provides an overview of our framework, which aims to optimize the allocation of EE DNNs on HMPSOCs to reduce latency variations by dividing the model into several execution graphs and running each graph on a separate processor. We implement the framework using the *ARM Compute Librart (ARM-CL)* library. Firstly, we outline the standard process for DNN inference within *ARM-CL* and then explain the adjustments we made to facilitate EE DNN parallelization.

ARM-CL Library: *ARM-CL* is a library of optimized ML functions for *ARM Cortex-A CPUs* and *Mali* GPUs. *ARM-CL* uses *NEON* and *Open Computing Library (OpenCL)* for

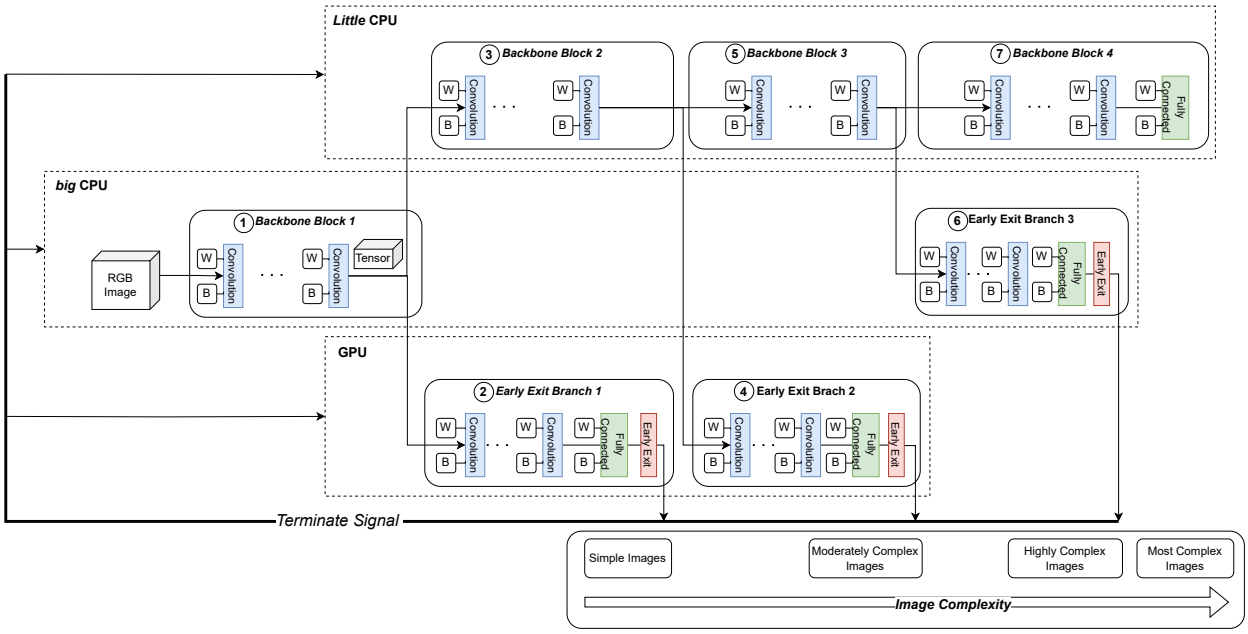

Fig. 2: An abstraction showing an overview of the distribution of exit branches and backbone blocks across various HMPSoC processors, allowing the exit branches to be processed independently without disrupting the backbone operations.

vectorized parallel computations on CPU and GPU, respectively. It represents DNNs as a graph, where the input and output layers map to Input and Output nodes. Hidden layers are defined by Main nodes, each linked to its own Weight and Bias nodes. The graph links all nodes in a chain sequence of consumer-producer connections. *ARM-CL* is initialized by creating a graph for the DNN model and setting up the CPU and GPU environment. For the CPU, it generates worker threads based on core count or user input. For the GPU, it gathers details such as core count and model and then creates an *OpenCL* context with an optimized scheduler for the detected device. *ARM-CL* determines tensor properties such as shape and data types from an input layer, allocates memory for weights and biases, and loads them with model parameters. It then serializes the kernels and prepares them for execution on the target processor. The input node first loads the image into memory to start the processing. For CPUs, computations split across worker threads, filling the output tensor with results. On GPUs, *ARM-CL* sends kernels to the *OpenCL* execute queue.

Cooperative Processor Allocation for Early Exits: In EE DNNs, when the model reaches an exit branch, it processes the branch and checks the exit policy, typically based on softmax confidence. However, halting the backbone model at each branch can increase latency for complex inputs, even surpassing that of static models. Unlike static models, where each layer depends on the previous layer output, EE DNNs only require the preceding backbone's output after each branch. Parallelizing the computation of exit branches eliminates the backbone waiting time during branch processing. Therefore, we distribute the backbone blocks and exit branches across all processors, allowing the model to continue processing the backbone on one processor while simultaneously evaluating the branch on the other.

We divide the original *ARM-CL* graph into sub-graphs to facilitate multi-processor inference. Therefore, each exit branch has its sub-graph and sub-graphs for the backbone blocks. New transfer and receiver nodes manage synchronized data transfer between these sub-graphs, allowing component switching during inference. The receiver node waits on the transfer node of the previous sub-graph by using a wait queue, freeing up computing resources. Once a sub-graph completes its process, its transfer node interrupts the corresponding receiver node to transfer data. Algorithm 1 provides the pseudocode for the framework.

Optimization: Each exit branch and backbone block in the model exhibits varying latencies across different processors. By redistributing these blocks and branches, we can adjust the best-, average-, and worst-case latency of EE DNNs. We first measure the latencies of all backbone blocks and exit branches on each processor to identify the optimal processor allocation for each block and exit branch. A Genetic Algorithm (GA) determines the best configuration using the collected data. The GA begins with a random allocation of processors to each block and exit branch, then iteratively refines the allocation to minimize average latency while ensuring that the worst-case latency remains below a specified target (e.g., the latency of the static model). Additionally, we account for the following system constraint during the search: the initialization of GPU is performed by big CPU. When big CPU and GPU run concurrently, this creates an overhead on the big CPU, which GA takes into account.

Algorithm 1 Cooperative Processor Allocation for Early Exits: Finding the optimal processor allocation using GA, creating subgraphs, and deploying subgraphs for edge inference

Initialization: Processors, EE_DNN
function ALLOCATE_PROCESSORS
  Input: EE_DNN, Processors
  $components\_latency$ = []
  for each $component$ in EE_DNN **do**
    for each $processor$ in Processors **do**
      l=latency($processor$, $component$)
      $components\_latency$.add(l)
    end for
  end for
  $optimal\_allocation$ = GA($components\_latency$)
  sub_graphs = []
  for each $component$ in $optimal\_allocation$ **do**
    if $component$.processor is changed **then**
      $graph$ = new sub_graph()
      $graph$.add($transfer\_node$)
      $graph$.add($receiver\_node$)
      sub_graphs.add($graph$)
    end if
    sub_graphs[-1].add($component$)
  end for
  return sub_graphs
end function
function DEPLOYMENT_ON_EDGE
  Input: sub_graphs, $thresholds$
  run_all(sub_graphs) // run all graphs in parallel
  for each sub_graph in sub_graphs **do**
    if sub_graph[0] is backbone **then**
      sub_graph.receive($feature\_map$)
    else if sub_graph[0] is exit_branch **then**
      transfer($feature\_map$) // to the next subgraph
    else if sub_graph.confidence $> threshold$ **then**
      return sub_graph.output
    end if
  end for
end function

Figure 2 illustrates a possible sub-graph structure for *ResNet18* with three exit branches, highlighting our core innovation. We process exit branches in parallel on the GPU and *big* CPU, independently of the backbone, allowing it to run without interruption. This parallel execution significantly reduces the end-to-end backbone's latency. Furthermore, if the branch output meets the criteria for early termination, a signal is sent to the backbone to terminate further processing, thereby reducing the extra computational cost.

IV. EXPERIMENTS

In this section, we first analyze three EE DNNs – *ResNet18*, *ResNet50*, and *VGG16* – along with their backbone models, assessing latency and accuracy for both in-distribution and out-of-distribution data. We then examine the best-, average-, and worst-case latencies of these DNNs across various HMPSoCs processors. Finally, we evaluate the impact of our framework on average- and worst-case inference latency, comparing it with that of static models.

TABLE I: Technical details for the experimental setup.

Experimental Setup	Details
Board	*Rock Pi N10*
HMPSoC	*RK3399PRO*
CPU	*ARM big.Little*
big CPU	Dual-core *ARM Cortex-A72*+
big CPU L1 cache	48 KB/32 KB I/D cache
big CPU L2 cache	1 MB
Little CPU	Quad-core *ARM Cortex-A53*
Little CPU L1 cache	32 KB/32 KB I/D cache
Little CPU L2 cache	512 KB
GPU	Quad-core *ARM Mali-T860 MP4* GPU
GPU L2 cache	256 KB
Main NPU	*Rockchip* NPU
Interconnect	Coherent Communication Interface 500 (CCI-500)
Memory	LPDDR3 3 GB (NPU 1 GB + CPU 2 GB)
OS	*Android 8.1*
Framework	*ARM-CO-UP* [24] based on *ARM-CL* v18.03

A. Experiment Setup

Models and Dataset. We employ three previously proposed EE DNNs: *ResNet18* [3], *ResNet50* [3], and *VGG16-BN* [6]. *ResNet18* and *ResNet50* incorporate three intermediate exit branches into the backbone model, featuring complex attention and scaling modules. In contrast, *VGG16-BN* has two intermediate exit branches, characterized by a more straightforward structure comprising three and two convolutional layers, respectively. All models are pre-trained on *CIFAR-100* [25], and we primarily focus on reducing latency while maintaining accuracy. We employ the *CIFAR100-C* [26] dataset for the performance evaluation with corrupted data. *CIFAR100-C* contains 18 corruption types: Gaussian Noise, Shot Noise, Impulse Noise, Defocus Blur, Glass Blur, Motion Blur, Zoom Blur, Snow, Frost, Brightness, Contrast, Elastic Transformation, Pixelate, JPEG Compression, Speckle Noise, Gaussian Blur, Spatter, and Saturate—each with five severity levels.

HMPSoC Setup. Table I details the *Rock Pi N10* embedded board used in this work. The platform features a hexa-core *ARM big.Little CPU* with asymmetric multi-core clusters: a quad-core *Little* CPU cluster containing four *Cortex-A53* cores with a maximum frequency of 1.4 GHz and a dual-core *big* CPU cluster with two *Cortex-A72* cores operating at 1.8 GHz. The HMPSoC includes a quad-core *ARM Mali-T860 MP4* GPU running at 0.8 GHz, with 3 GB of LPDDR3 main memory allocated for the CPU clusters and GPU. The platform runs on *Android v8.1*, and on top of that, we use *ARM-CO-UP* [24], based on *ARM-CL 18.03*, for DNN inference in this work. Our methodology builds on *ARM-CO-UP*'s existing capabilities while integrating additional functionalities tailored to this study. Furthermore, even though we present the results

TABLE II: Latency and accuracy trade-off evaluation of *ResNet18*, *ResNet50*, and *VGG16* EE DNNs at different confidence thresholds, including the percentage of samples classified at each exit on *CIFAR-100*.

Models	**Confidence Threshold**	**Classifiers' Utility (%)**				**Accuracy**
		1	**2**	**3**	**4**	
ResNet18 [3]	1.0	0	0	0	100	81.76
	0.9	55.09	10.02	7.58	27.31	81.48
	0.8	64.28	10.27	7.30	18.15	81.04
	0.6	77.67	9.95	5.76	6.62	79.97
	0.4	90.49	5.95	2.16	1.40	78.03
ResNet50 [3]	1.0	0	0	0	100	83.66
	0.9	68.86	6.79	3.51	20.84	83.53
	0.8	75.94	7.08	3.69	13.29	83.86
	0.6	86.16	6.17	2.50	5.17	83.56
	0.4	94.77	3.28	1.01	0.94	82.68
VGG16-BN [6]	1.0	0	0	100	-	73.02
	0.9	1.54	10.90	87.56	-	72.97
	0.8	3.38	16.30	80.32	-	72.62
	0.6	9.08	26.93	63.99	-	70.39
	0.4	20.03	37.58	42.39	-	64.00

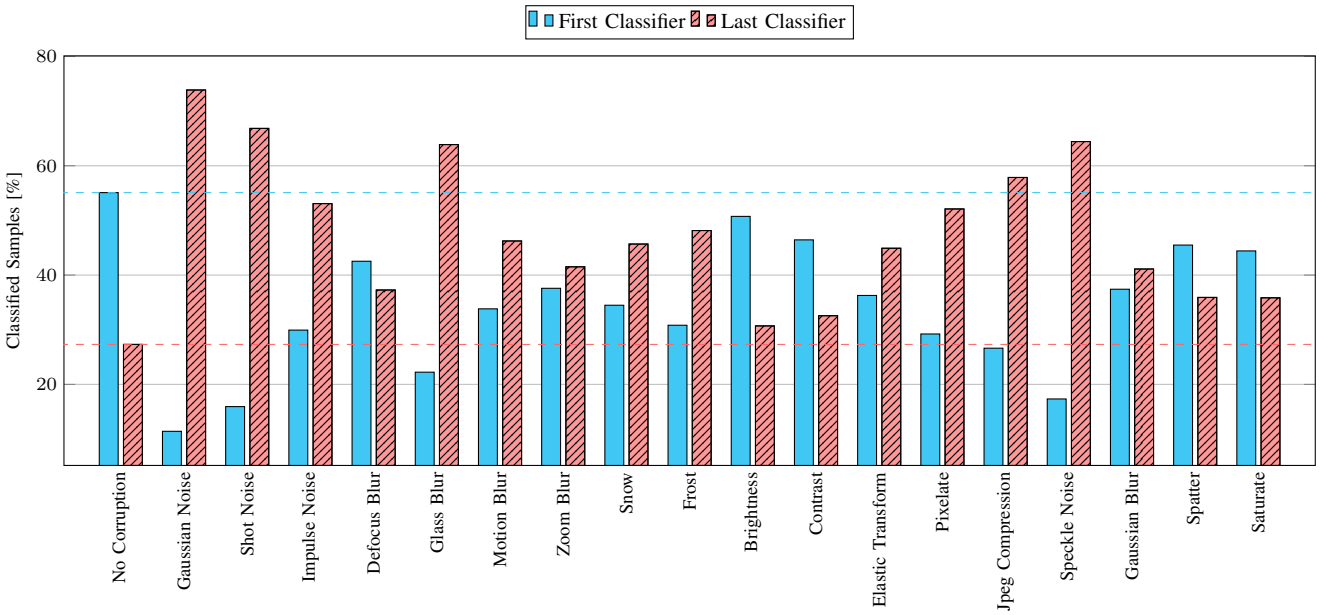

Fig. 3: Percentage of samples classified by the first and the last classifiers in the EE *ResNet18* model on *CIFAR100-C*, with a confidence threshold of 0.9. The dataset includes 18 corruption types, each with five severity levels.

on one specific board, the proposed methodology applies to conventional embedded devices wherein the GPUs and CPUs have comparable performance.

GA Setup. We employ a GA [27] for meta-heuristic optimization with the following parameters: a population size of 500, 250 generations, crossover probability of 0.8, and mutation probability of 0.5. The fitness function includes average-case (objective) and worst-case (constraint) latencies.

Profiling Time. The Graph Manager component in *ARM-CO-UP* [24] has a profiling feature that reports the average execution time across all frames upon request. It also monitors timing for inter-processor communication, input, and output operations. The best- and worst-case latency corresponds to the latency of the first exit branch and the final classifier, respectively. We calculate the average latency by multiplying the percentage of samples classified by each classifier by the latency of the corresponding exit branch, summing these values, and then dividing by the total number of test samples. We calculate the speed-up by dividing the EE DNN's average-case latency by the latency of the static model.

B. Trade-off between Latency and Accuracy

The trade-off between latency and accuracy in EE DNNs depends on factors such as the architecture of the exit branches and the exit policy based on the confidence of the predicted label. Table II analyzes three EE DNNs: *ResNet18*, *ResNet50* [3] (with complex exit branches), and *VGG16* [6] (with a simple exit branch). The exit policy relies on a confidence

TABLE III: Comparison of static model latency with the best-, average-, and worst-case latency of EE *ResNet18*, *ResNet50*, and *VGG16* when processing on a single processor. Latency overhead indicates the increase in latency of EE DNNs compared to static models when input data is complex. The models are trained on *CIFAR-100*, and the confidence threshold during inference is set to 0.6.

Model	Processor	Static Model Latency (ms)	EE Model Latency (ms)			Latency Overhead for
			Best	Average	Worst	Complex Samples (ms)
ResNet18	*big* CPU	36.91	16.24	21.12	**52.04**	**15.13 (37.9%)**
	Little CPU	37.10	23.66	28.69	**60.78**	**23.68 (63.8%)**
	GPU	**36.62**	13.86	**18.60**	**48.75**	**24.08 (33.12%)**
ResNet50	*big* CPU	180.01	79.89	**97.98**	**272.59**	**92.58 (51%)**
	Little CPU	**160.11**	90.59	107.88	**273.59**	**113.48 (70.4%)**
	GPU	204.29	76.8	98.911	**320.55**	**116.26 (56.9%)**
VGG16-BN	*big* CPU	62.24	3.48	**45.67**	**66.42**	**4.18 (6%)**
	Little CPU	60.97	4.54	45.84	**65.93**	**4.96 (8%)**
	GPU	**56.83**	6.49	46.50	**65.68**	**8.85 (15.5%)**

threshold. If the confidence of a sample in an exit branch exceeds this threshold, it will exit at that branch. We varied the threshold from 1 (where all exits are disabled) to 0.4, measuring the percentage of samples exiting at each branch. Lowering the threshold allows more samples to exit earlier, reducing latency but potentially lowering accuracy. For *ResNet* models, with their complex exit branches, more than 90% of the samples exit through the first exit branch using a lower threshold, resulting in a significant latency reduction with minimal accuracy drop. However, in *VGG16*, where exit branches have only two or three convolutional layers, most samples exit at the last classifier, and lowering the threshold leads to an accuracy drop.

Performance of Exit Branches with Corrupted Data. The latency of EE DNNs depends on the complexity of the input samples. As shown in Table II for *ResNet18* and *ResNet50* with a threshold of 0.6, fewer than 10% of samples must be classified by the last classifier, which indicates they are considered complex samples. However, since the first exit branch has fewer layers than the backbone model, its generalizability is limited. The performance of the exit branch can decrease when the data distribution of the samples changes, resulting in more samples needing to be processed by the final classifier.

Figure 3 illustrates the performance of the first and last exit branches of *ResNet18* with a confidence threshold of 0.9. It shows the percentage of samples classified by the initial and final classifiers when testing the model on standard inputs (*CIFAR100*) and corrupted inputs (*CIFAR100-C*) with various types of corruption. Figure 3 shows that 55% of the samples are classified by the first exit branch when there is no corruption. However, this percentage significantly drops for corrupted samples, with the final classifier processing more samples. Specifically, for certain types of corruption, such as Gaussian noise, the percentage of samples classified by the first exit branch decreases sharply to less than 10%. These results demonstrate that when more samples are out of distribution, the latency reduction with an EE DNN can be greatly reduced since more samples are considered complicated These samples also lead to a substantial latency overhead, as all exit branches must process them, in addition to the backbone and final classifier. This overhead leads to a significant increase in overall inference latency compared to static DNNs.

C. EE DNNs on Single Processor

Table III presents the latency of static models alongside the best- (first exit), average-, and worst-case (last exit) latencies for EE *ResNet18*, *ResNet50*, and *VGG16*, compared to the static models when processed on single processors. A comparison of static models latencies across different processors shows that *ResNet50* had lower latency on the CPU, while *ResNet18* and *VGG16* exhibited lower latency on the GPU. For EE DNNs, EE *ResNet18* demonstrated lower average latency on the GPU, while EE *ResNet50* and EE *VGG16* exhibited lower average latency on the CPU.

Comparing the static model latency with the best- and average-case latencies of EE DNNs demonstrates that EE DNNs effectively reduce average latency. However, considering the worst-case (latency for more complex samples), the exit branches introduce a latency overhead for these cases. The latency overhead for different target processors varies from 33.12% to 63.8%, 51% to 70.4%, and 6% to 15.5% for *ResNet18*, *ResNet50*, and *VGG16*, respectively.

Our results indicate that the conventional strategy of deploying EE DNNs across different processors can lead to varying latency. However, latency variations can be significant, often surpassing the static model. This variation is especially noticeable when the exit branches are more complex, resulting in a considerable increase in latency overhead. For instance, the latency overhead for the EE *ResNet50* was approximately 100 ms higher than the static model. This overhead is problematic for applications where low latency or latency consistency is crucial.

TABLE IV: Comparison of static model latency(with the best processor) with the best-, average-, and worst-case latency of EE *ResNet18*, *ResNet50*, and *VGG16* when processing the backbone and exit branches in parallel on different processors. Latency overhead represents the increase in latency of EE DNNs compared to static models for complex input data. Speed-up is calculated by dividing the EE model's average latency, measured on the best single processor and after deployment on multiple processors, by the latency of the static model. The models are trained on *CIFAR-100*, with a confidence threshold of 0.6 during inference.

Model	**Static Model Latency (ms)**	**EE DNN Speed-up on Single Processor**	**EE DNN on Multi processors**					
			Processors	**Best Latency**	**Average Latency**	**Worst Latency**	**Latency Overhead**	**Speed-up**
ResNet18	36.62 (G)	1.96 (G)	BGLGLBL	15.3	**18.12**	**35.3**	**-0.32 (0%)**	**2.01**
ResNet50	160.11 (L)	1.63 (B)	BGLBLBL	79.21	**89.05**	**181.66**	**21.55 (13.4%)**	**1.79**
VGG16-BN	56.83 (G)	1.24 (B)	BLBLG	5.94	**40.36**	**58.03**	**1.2 (2.11%)**	**1.40**

D. EE DNNs on HMPSoCs

This paper strives to minimize latency variations, particularly for more complex samples, thereby reducing the average- and worst-case latency of EE DNNs. Our framework chooses the most appropriate processor for each exit branch and backbone block, allowing for the parallel execution of exit branches alongside the backbone based on their execution times, as detailed in Section III. Table IV compares the static model latency (when running the backbone on the best processor) with the best-, average-, and worst-case latency of EE *ResNet18*, *ResNet50*, and *VGG16* at a 0.6 confidence threshold under this configuration. Additionally, the speed-up is determined by dividing the EE model's average latency—measured on the best single processor and after deployment across multiple processors (with the optimal processor for each component determined by GA)—by the lowest latency of the static model.

For EE *ResNet18*, the results showed that our method reduce both the average- and worst-case inference latency compared to the model running on a single processor (see Table III). As a result, the latency overhead decreased by over 33%, reaching 35.3 ms, lower than the static model. Furthermore, for static *ResNet50*, the lowest latency was observed on the *Little CPU*. Comparing the static latency with the worst-case latency in EE *ResNet50* revealed that the latency overhead was significantly reduced by 37.6%, along with a decreased average latency. Similarly, for EE *VGG16* with two simple exit branches, we see a reduction in both the average- and worst-case latency. Compared to the lowest inference latency of the static model running on GPU, the latency overhead decreases by 3.89%.

Comparing the latency of EE DNNs on multiple processors to that on a single processor demonstrated that, across all models, both the average- and worst-case latencies were reduced relative to the best single-processor setup. This decrease in latency led to a slight improvement in speed-up, as shown in Tables III and IV. The results show that deploying EE DNNs, unlike the static DNNs, on an HMPSOC can cause substantial latency variations, which may limit their full potential. However, leveraging the diverse processors within an HMPSoC can effectively minimize these variations, making them highly beneficial for latency-sensitive applications.

We primarily focused on EE DNNs, a common type of dynamic model. However, our framework is adaptable and can facilitate the deployment of other dynamic networks, such as skipping layers [28] and dynamic routing [10] on HMPSoC platforms. For example, in dynamic routing, routes can be replaced with exit branches, while in skipping layers, skip connections can be treated as exit branches within the framework. This generates subgraphs for each route or skip branch, enabling, multiple routes to execute and communicate in parallel efficiently. This approach can improve model speed-up and reduce overall latency while ensuring the worst-case latency remains within the bounds of static models. Furthermore, since there are no modifications to the exit policies or the model's parameters and weight values, this framework does not impact the model's accuracy. Therefore, we can apply it to pre-trained EE DNNs before deployment on HMPSOCs.

V. Conclusion And Future Work

We examined the challenge of latency variations in EE DNNs, which can complicate their deployment in real-world applications. We demonstrated that the waiting time caused by exit branches can increase the latency of complex samples beyond that of static models. Our findings showed that the latency of different processors in HMPSoC is similar. Therefore, we can utilize multiple HMPSoC processors to reduce this waiting time. We reduced the worst-case latency to 35.3 ms for *ResNet18*, 181.66 ms for *ResNet50*, and 58.03 ms for *VGG16* by modifying the computation graph and distributing the EE DNNs across different processors. This approach reduces latency variations in EE DNNs, achieving a modest improvement in speed-up over single-processor deployment while preserving model performance.

We explored the impact of a widely used dynamic model on HMPSoC devices. Future works could extend this investigation by optimizing these dynamic models for other tasks, such as object detection on edge devices, to reduce overall latency. Moreover, instance-aware model compression techniques, such as instance-aware [29] or partial [30] quantization inference, which adaptively compress the model based on input data during inference, present another promising direction for exploration on HMPSoC devices. These models can leverage

multiple processors to provide consistent latency for all sample types and facilitate the deployment of deep learning models on edge devices, especially the larger models.

REFERENCES

[1] R. Singh and S. S. Gill, "Edge ai: a survey," *Internet of Things and Cyber-Physical Systems*, vol. 3, pp. 71–92, 2023.

[2] D. Liu, H. Kong, X. Luo, W. Liu, and R. Subramaniam, "Bringing ai to edge: From deep learning's perspective," *Neurocomputing*, vol. 485, pp. 297–320, 2022.

[3] L. Zhang, C. Bao, and K. Ma, "Self-distillation: Towards efficient and compact neural networks," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 44, no. 8, pp. 4388–4403, 2021.

[4] S. Teerapittayanon, B. McDanel, and H.-T. Kung, "Branchynet: Fast inference via early exiting from deep neural networks," in *2016 23rd international conference on pattern recognition (ICPR)*. IEEE, 2016, pp. 2464–2469.

[5] S. Wang, A. Pathania, and T. Mitra, "Neural network inference on mobile socs," *IEEE Design & Test*, vol. 37, no. 5, pp. 50–57, 2020.

[6] R. Dong, Y. Mao, and J. Zhang, "Resource-constrained edge ai with early exit prediction," *Journal of Communications and Information Networks*, vol. 7, no. 2, pp. 122–134, 2022.

[7] D. Hendrycks, "CIFAR-100-C," Jan. 2019. [Online]. Available: https://doi.org/10.5281/zenodo.3555552

[8] Y. Han, G. Huang, S. Song, L. Yang, H. Wang, and Y. Wang, "Dynamic neural networks: A survey," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 44, no. 11, pp. 7436–7456, 2021.

[9] J. Lin, Y. Rao, J. Lu, and J. Zhou, "Runtime neural pruning," *Advances in neural information processing systems*, vol. 30, 2017.

[10] H. Hazimeh, N. Ponomareva, P. Mol, Z. Tan, and R. Mazumder, "The tree ensemble layer: Differentiability meets conditional computation," in *International Conference on Machine Learning*. PMLR, 2020, pp. 4138–4148.

[11] B. Yang, G. Bender, Q. V. Le, and J. Ngiam, "Condconv: Conditionally parameterized convolutions for efficient inference," *Advances in neural information processing systems*, vol. 32, 2019.

[12] S. Disabato and M. Roveri, "Reducing the computation load of convolutional neural networks through gate classification," in *2018 International Joint Conference on Neural Networks (IJCNN)*, 2018, pp. 1–8.

[13] J. Pomponi, S. Scardapane, and A. Uncini, "A probabilistic re-intepretation of confidence scores in multi-exit models," *Entropy*, vol. 24, no. 1, 2022. [Online]. Available: https://www.mdpi.com/1099-4300/24/1/1

[14] T. Bolukbasi, J. Wang, O. Dekel, and V. Saligrama, "Adaptive neural networks for efficient inference," in *Proceedings of the 34th International Conference on Machine Learning*, ser. Proceedings of Machine Learning Research, D. Precup and Y. W. Teh, Eds., vol. 70. PMLR, 06–11 Aug 2017, pp. 527–536. [Online]. Available: https://proceedings.mlr.press/v70/bolukbasi17a.html

[15] M. Odema, N. Rashid, and M. A. Al Faruque, "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, pp. 1–6.

[16] M. Ayyat, T. Nadeem, and B. Krawczyk, "Classynet: Class-aware early exit neural networks for edge devices," *IEEE Internet of Things Journal*, pp. 1–1, 2023.

[17] M. Gambella and M. Roveri, "Edanas: Adaptive neural architecture search for early exit neural networks," in *2023 International Joint Conference on Neural Networks (IJCNN)*. IEEE, 2023, pp. 1–8.

[18] F. Ilhan, K.-H. Chow, S. Hu, T. Huang, S. Tekin, W. Wei, Y. Wu, M. Lee, R. Kompella, H. Latapie, G. Liu, and L. Liu, "Adaptive deep neural network inference optimization with eenet," in *Proceedings of the IEEE/CVF Winter Conference on Applications of Computer Vision (WACV)*, January 2024, pp. 1373–1382.

[19] S. Wang, G. Ananthanarayanan, Y. Zeng, N. Goel, A. Pathania, and T. Mitra, "High-throughput cnn inference on embedded arm big. little multicore processors," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 39, no. 10, pp. 2254–2267, 2019.

[20] E. Jeong, J. Kim, S. Tan, J. Lee, and S. Ha, "Deep learning inference parallelization on heterogeneous processors with tensorrt," *IEEE Embedded Systems Letters*, vol. 14, no. 1, pp. 15–18, 2021.

[21] E. Aghapour, D. Sapra, A. Pimentel, and A. Pathania, "Cpu-gpu layer-switched low latency cnn inference," in *Proc. of the Euromicro Conference on Digital System Design (DSD)*, 2022, pp. 324–331.

[22] A. Karatzas and I. Anagnostopoulos, "Omniboost: Boosting throughput of heterogeneous embedded devices under multi-dnn workload," in *2023 60th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 2023, pp. 1–6.

[23] A. Prakash, N. Ramakrishnan, K. Garg, and T. Srikanthan, "Accelerating computer vision algorithms on heterogeneous edge computing platforms," in *2020 IEEE Workshop on Signal Processing Systems (SiPS)*. IEEE, 2020, pp. 1–6.

[24] E. Aghapour, D. Sapra, A. Pimentel, and A. Pathania, "Arm-co-up: Arm cooperative utilization of processors," *ACM Transactions on Design Automation of Electronic Systems*, vol. 29, no. 5, 2024.

[25] A. Krizhevsky, G. Hinton *et al.*, "Learning multiple layers of features from tiny images," 2009.

[26] D. Hendrycks and T. Dietterich, "Benchmarking neural network robustness to common corruptions and perturbations," *arXiv preprint arXiv:1903.12261*, 2019.

[27] F.-A. Fortin, F.-M. De Rainville, M.-A. Gardner, M. Parizeau, and C. Gagné, "DEAP: Evolutionary algorithms made easy," *Journal of Machine Learning Research*, vol. 13, pp. 2171–2175, jul 2012.

[28] X. Wang, F. Yu, Z.-Y. Dou, T. Darrell, and J. E. Gonzalez, "Skipnet: Learning dynamic routing in convolutional networks," in *Proceedings of the European conference on computer vision (ECCV)*, 2018, pp. 409–424.

[29] Z. Liu, Y. Wang, K. Han, S. Ma, and W. Gao, "Instance-aware dynamic neural network quantization," in *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, 2022, pp. 12 434–12 443.

[30] E. Aghapour, Y. Shen, D. Sapra, A. Pimentel, and A. Pathania, "Piqi: Partially quantized dnn inference on hmpsocs," in *Proceedings of the 29th ACM/IEEE International Symposium on Low Power Electronics and Design*, 2024, pp. 1–6.