IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received October 16, 2020, accepted November 13, 2020, date of publication November 18, 2020,
date of current version December 11, 2020.

Digital Object Identifier 10.1109/ACCESS.2020.3038908

Energy-Efficient Acceleration of Deep Neural Networks on Realtime-Constrained Embedded Edge Devices

BOGIL KIM[1], SUNGJAE LEE[1], AMIT RANJAN TRIVEDI[2], (Member, IEEE), AND WILLIAM J. SONG[1], (Member, IEEE)

[1]School of Electrical and Electronic Engineering, Yonsei University, Seoul 03722, South Korea
[2]Department of Electrical and Computer Engineering, University of Illinois at Chicago, Chicago, IL 60607 USA

Corresponding author: William J. Song (wjhsong@yonsei.ac.kr)

This work was supported by the Ministry of Trade, Industry and Energy and Korea Semiconductor Research Consortium under Grant #10080674, and in part by the Ministry of Science and ICT, South Korea, through the ITRC-supported program supervised by the Institute of Information and Communications Technology Planning and Evaluation under Grant #2020-0-01847.

ABSTRACT This paper presents a hardware management technique that enables energy-efficient acceleration of deep neural networks (DNNs) on realtime-constrained embedded edge devices. It becomes increasingly common for edge devices to incorporate dedicated hardware accelerators for neural processing. The execution of neural accelerators in general follows a host-device model, where CPUs offload neural computations (e.g., matrix and vector calculations) to the accelerators for datapath-optimized executions. Such a serialized execution is simple to implement and manage, but it is wasteful for the resource-limited edge devices to exercise only a single type of processing unit in a discrete execution phase. This paper presents a hardware management technique named *NeuroPipe* that utilizes heterogeneous processing units in an embedded edge device to accelerate DNNs in energy-efficient manner. In particular, NeuroPipe splits a neural network into groups of consecutive layers and pipelines their executions using different types of processing units. The proposed technique offers several advantages to accelerate DNN inference in the embedded edge device. It enables the embedded processor to operate at lower voltage and frequency to enhance energy efficiency while delivering the same performance as uncontrolled baseline executions, or inversely it can dispatch faster inferences at the same energy consumption. Our measurement-driven experiments based on NVIDIA Jetson AGX Xavier with 64 tensor cores and eight-core ARM CPU demonstrate that NeuroPipe reduces energy consumption by 11.4% on average without performance degradation, or it can achieve 30.5% greater performance for the same energy consumption.

INDEX TERMS Deep neural networks, heterogeneous computing, embedded processors, hardware management, hardware measurement, energy efficiency, performance.

I. INTRODUCTION

Deep neural networks (DNNs) have become important applications in diverse domains encompassing autonomous driving, surveillance cameras, and a variety of Internet of Things (IoT) gadgets. Deep learning on edge devices has potentially many advantages such as security and latency. However, deploying DNN workloads on embedded systems imposes great hardware burdens on the edge devices around performance, energy, power, and thermal. It is challenging to perform realtime-constrained inference of DNNs on the resource-limited embedded edge devices in energy-efficient manner for their massive amount of operations and sizable data. To amortize the computational costs of DNNs, recent approaches seek to design lighter neural networks such as MobileNet [9], [25] and ShuffleNet [31] by engaging novel calculation methods that reduce the amount of operations (e.g., depth-wise convolution [10], point-wise group convolution [31]). Other directions include devising optimization techniques such as compression [6], pruning [7], [14], quantization [11], [29], and mixed precision [16], [26].

In addition to the continued efforts to make neural network computations more efficient and affordable, it becomes increasingly common in embedded edge devices

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

TABLE 1. Examples of CPU and neural accelerator integration in embedded edge devices [1], [4], [18], [27].

Edge device		NVIDIA Jetson TX2	NVIDIA AGX Xavier	Intel Myriad X VPU	Tesla FSD	Google Edge TPU
CPU	Architecture	ARM Cortex-A57	ARM v8.2	SPARC v8 RISC	ARM Cortex-A72	ARM Cortex-A53
	# of PUs	4	8	2	12	4
Accelerator	Type	GPU	GPU, tensor cores	VLIW vector processors	Neural processing units	Systolic arrays
	# of PUs	256 CUDA cores	512 CUDA, 64 tensor cores	16 SHAVE cores	96×96 MACs	128×128 MXU

to incorporate dedicated hardware accelerators for neural processing. The accelerators as specialized processing units provide neural applications with faster and more power-efficient computing solutions. Exemplary neural network accelerators found in embedded edge devices encompass the tensor cores of NVIDIA Jetson AGX Xavier GPU [4], VLIW vector processors in Intel Myriad X VPU [18], multiply-accumulate (MAC) units of Tesla FSD chip [27], and systolic arrays of Google Edge TPU [1]. The execution of a neural accelerator typically follows a host-device model, where a host CPU initiates and then offloads a series of neural computation kernels of an application to the device accelerator that implements the optimized datapath of neural networks. As illustrated in Figure 1a, the conventional host-device execution method (i.e., alternating executions between the CPU and accelerator) is simple to implement and manage, but it is wasteful for the resource-limited embedded edge device to exercise only a single type of processing unit (PU) in every discrete execution phase.

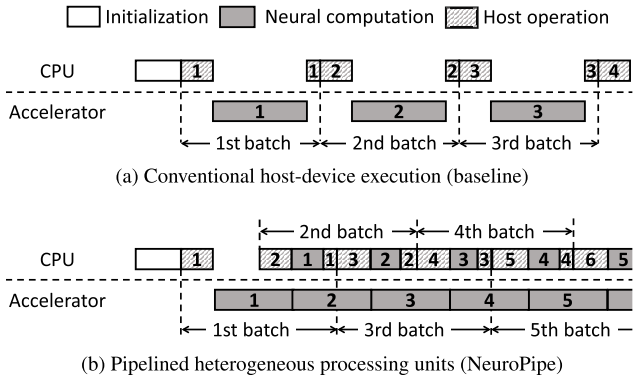


FIGURE 1. (a) Conventional execution of heterogeneous PUs, where different types of PUs are activated at discrete phases. (b) Pipelined executions of heterogeneous PUs cooperatively accelerate neural computations, thereby delivering greater throughput in a resource-limited edge device.

Notably, neural accelerators are not standalone processing units but supplementary components in embedded processors to aid the execution of neural network applications. As summarized in Table 1, neural accelerators found in commercially-available edge devices are integrated with a few to many high-performance CPU cores. The CPU cores are used for handling host interfaces, system services, and other irregular operations. However, they do not engage in actual neural processing while the accelerators perform offloaded neural computations, based on the conventional host-device execution model illustrated in Figure 1a. Consequently, the CPU cores are left idle and wasted for a large fraction of execution time, and the resource-limited embedded edge devices miss valuable opportunities to draw additional performance. The extra performance may be harvested by escalating the operating voltage and frequency of neural processing units, but such an approach in turn deteriorates energy efficiency and exacerbates power and thermal concerns in the edge devices. Instead of activating only a single type of processing unit, utilizing different kinds of PUs including both neural accelerators and CPU cores in the resource-limited embedded edge devices can deliver additional throughput for neural processing.

This paper presents a novel hardware management technique named *NeuroPipe* that enables cooperative acceleration of heterogeneous PUs in an embedded edge device for faster neural processing in energy-efficient manner. Experiments based on hardware measurements demonstrate later in the paper that the proposed technique not only improves performance but also saves total energy consumption. In particular, the proposed NeuroPipe technique splits a neural network into groups of consecutive layers and executes them using different types of PUs in pipelined fashion as shown in Figure 1b. The CPU reads an incoming input and offloads its neural computations to an accelerator. The accelerator executes neural layers in a lazy way such that it does not attempt to finish the entire neural computations of the given batch input (e.g., 1st batch) before the next batch input arrives. The unfinished part of neural network is handed over to CPU cores, and the neural accelerator simply moves on to the newly arrived input (e.g., 2nd batch). While the accelerator handles the neural processing of new input, the CPU executes the remaining part of previous batch's neural computations and finalizes the inference. As soon as another input arrives (e.g., 3rd batch), the CPU reads the input and makes it ready to be offloaded to the accelerator. Then, it takes over the incomplete neural computations of preceding batch from the accelerator. As described, NeuroPipe makes the inferences of streaming inputs overlap in pipelined manner, and it can achieve greater throughput than the conventional host-device execution method. In addition, NeuroPipe reduces the total energy consumption while delivering greater throughput by eliminating the idle cycles of neural accelerator and CPU cores. Thus, the proposed NeuroPipe technique significantly enhances the energy efficiency of embedded edge device for neural processing.

A workload scheduler of NeuroPipe profiles runtime characteristics of a neural application and splits the neural network into heterogeneous PUs in pipelined fashion.

The scheduler leverages a simple yet accurate theoretical model that estimates the achievable performance speedup of NeuroPipe and finds an optimal workload partitioning point. By pipelining the neural executions of streaming inputs, NeuroPipe has an effect of relaxing a realtime constraint in a resource-limited embedded edge device. In particular, a neural accelerator in the edge device is not obligated to satisfy the realtime constraint of neural processing on its own. An incomplete part of neural network is handed over to high-performance CPU cores integrated in the same embedded processor package, and the accelerator simply moves on to the next batch of input. Thus, the edge device can deliver greater performance (e.g., inferences per second) via cooperative executions between the heterogeneous PUs, which otherwise should be fulfilled by increasing the operating voltage and frequency to produce the desired throughput.

The proposed NeuroPipe technique opens up hardware management opportunities in a resource-limited embedded edge device. In case the performance is not a limiting factor for the operations of edge device, it can exploit the aforementioned advantages to reduce the operating voltage and frequency of embedded processor and minimize the total energy consumption while generating the same computational throughput as the baseline execution method (i.e., conventional host-device execution). On the contrary, if greater throughput is needed to meet a realtime constraint, NeuroPipe can boost up the operating voltage and frequency for faster inference while consuming the same amount of energy as the baseline execution. We demonstrate such management capabilities later in the paper based on hardware measurements of NVIDIA Jetson AGX Xaiver [4] with eight ARM CPU cores and 64 tensor cores as exemplary hardware. Importantly, the proposed NeuroPipe technique is not limited only to CPU-accelerator executions in embedded edge devices but applicable to a wider range of devices leveraging the host-device execution model.

The remainder of this paper is organized as follows. We first review related work regarding neural network acceleration via workload scheduling, and we discuss the shortcomings of prior techniques if they were applied to embedded edge devices. Then, we present our methodology to pipeline neural network applications via cooperative executions between heterogeneous PUs. Lastly, the performance, power, energy, and thermal implications of proposed NeuroPipe technique are presented based on hardware measurements of an embedded edge device.

II. RELATED WORK

Cooperative executions of heterogeneous PUs have been explored in various prior studies as summarized in the survey of Mittal and Vetter [17]. In this section, we focus on the related work regarding neural network computations since different kinds of workloads pose different computing challenges. Neural acceleration techniques using heterogeneous PUs can be largely categorized into two approaches, i) inner-layer slicing and ii) network pipelining as depicted in Figure 2.

The most common approach found in the related work leverages inner-layer slicing and parallelization, where different types of processing units simultaneously take part in executing a neural layer by dividing it in proportion to their performance difference as shown in Figure 2a. For example, Zhong *et al.* [32] proposed a load balancing method to distribute high-performance computing (HPC) workloads to CPUs and GPUs, and Ohshima *et al.* [20] used both CPUs and GPUs to speed up matrix multiplication. However, when this kind of methodology is applied to neural network computations in an embedded edge device, it generates a substantial communication overhead between heterogeneous PUs at the boundary (i.e., barrier) of every neural layer to synchronize computation results. Consequently, such an approach significantly penalizes the overall performance. It also requires the heterogeneous PUs to maintain duplicate copies of neural data for them to work in parallel, which is not affordable in the resource-limited embedded edge device.

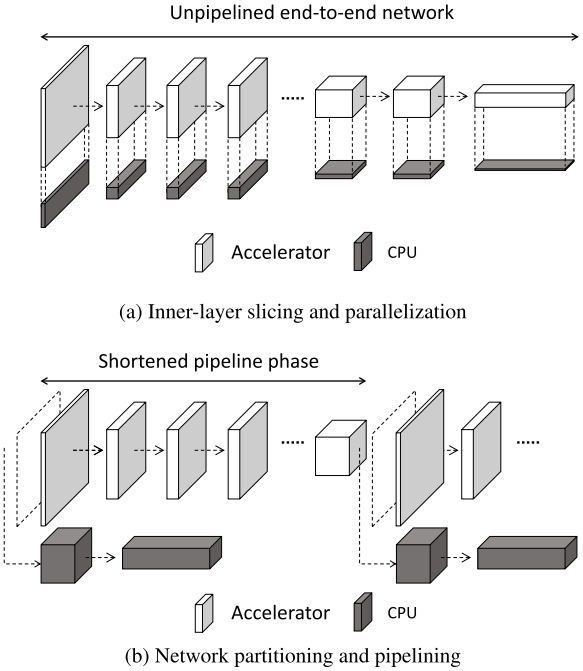


FIGURE 2. **(a) Each layer is split and distributed to heterogeneous PUs in proportion to their computing capabilities. It generates a substantial communication overhead to synchronize neural data at every layer. (b) A neural network is partitioned into groups of consecutive layers, and they are executed by different types of PUs in pipelined manner.**

Table 2 shows how the inner-layer slicing method compares to the baseline execution and proposed NeuroPipe technique. The experiment was conducted with MobileNet [25] as an exemplary DNN workload, and it was executed by a combination of eight ARM CPU cores and 64 tensor cores in NVIDIA Jetson AGX Xavier [4]. The table shows the execution time (i.e., inference interval) of three different schemes and time breakdown into neural computation and

TABLE 2. Comparison of execution methods for MobileNet.

Execution method	Baseline	Slicing	Pipelining (NeuroPipe)
Inference interval	40.7ms	45.9ms	28.0ms
Neural computation	29.3ms	28.5ms	31.2ms
Communication overhead	11.4ms	17.4ms	12.1ms
Memory requirement	161.5MB	285.7MB	161.5MB

TABLE 3. Comparison of pipelining methods for ResNet.

Execution method		Baseline	PU-oblivious	PU-aware (NeuroPipe)
Inference interval		67.9ms	61.8ms	53.1ms
Host operation (CPU)		11.9ms	14.2ms	13.4ms
Neural computation	CPU	-	47.6ms	30.8ms
	Accelerator	56.0ms	49.7ms	53.1ms

data communication portions. The last row shows the memory usage to implement the execution schemes. The results reveal that the inner-layer slicing method in the edge device rather degrades the performance because the layer-by-layer communication overhead subdues the neural computation speedup. It also uses a significantly larger memory space than other schemes since the heterogeneous PUs need to retain separate copies of neural data to proceed in parallel. In contrast, the network partitioning and pipelining method (i.e., NeuroPipe) following an execution sequence shown in Figure 2b effectively reduces the runtime with no additional memory space requirements compared with the baseline. Although NeuroPipe slightly prolongs the neural computation time because of parallelization overhead, pipelined executions of neural layers eventually lead to overall performance speedup. Since the heterogeneous PUs work on disjoint parts of the neural network, this method does not require replicating data across the PUs for parallelization and thus has the same memory footprint as the baseline execution.

There were found a few related efforts that exploited the pipelining method illustrated in Figure 2b. This approach splits a workload into pipelined phases and makes their executions overlap using multiple processing units. For instance, Yang *et al.* [30] presented a pipelined execution method to increase system throughput for computer vision applications using multiple GPUs. PipeDream [19] attempted to exploit both inner-layer slicing and network pipelining methods using multiple GPUs in an HPC environment for neural network training. However, these multi-GPU schemes are not applicable to accelerating DNN inference in a resource-limited embedded edge device.

Pipe-it [28] proposed a multi-stage pipelining scheme for convolutional neural networks in a mobile processor comprised of ARM big.LITTLE CPUs. It utilizes the asymmetric CPU cores for neural processing, but its assumed environment has become outdated in that recent mobile or embedded devices incorporate dedicated neural accelerators [21], [22]. In addition, utilizing heterogeneous PUs in an edge device addresses a different scheduling challenge than simply engaging only the general-purpose CPUs. When a Pipe-it-like approach is applied to ResNet [8] using eight ARM CPU cores and 64 tensor cores in NVIDIA Jetson AGX Xavier [4], it fails to find an optimal partitioning point for pipelined executions as shown in Table 3. The PU-oblivious methodology partitions the neural computations by considering only the performance difference between the PUs but not the different roles of distinct PU types (i.e., host and device). Hence, the CPU cores become overloaded with inherent host operations and excessive number of neural layers that lead to a longer inference interval for non-optimal partitioning. On the contrary, NeuroPipe distinguishes host operations and neural computations by following an execution order illustrated in Figure 1b, and it finds the optimal partitioning point and scheduling sequence for the heterogeneous PUs such that the CPU cores do not become a bottleneck due to overloaded neural computations. Therefore, the proposed NeuroPipe technique is better suited for neural executions in the embedded edge device and shows superior performance to the prior methods. In the remainder of this paper, we focus and elaborate on the details of NeuroPipe scheme and its evaluation based on hardware measurements.

III. METHODOLOGY

A. MOTIVATION AND OPPORTUNITIES

The conventional host-device execution model shown in Figure 1a exercises only a single type of processing unit in a discrete execution phase. Such an approach leaves a neural accelerator and CPU in an embedded edge device idle for a significant fraction of execution time. For instance, executing MobileNet in NVIDIA Jetson AGX Xavier makes CPU cores and accelerator become idle for 71% and 29% of runtime, respectively. The underutilized compute resources miss opportunities to draw additional performance in the resource-limited edge device. This observation becomes a motivation to develop the NeuroPipe technique that minimizes the idle cycles of compute resources via load balancing between the heterogeneous PUs in pipelined manner.

The proposed methodology for pipelined neural executions of heterogeneous PUs in an embedded edge device follows an execution sequence illustrated in Figure 1b. The CPU reads an incoming input (i.e., pre-processing) and then offloads neural computations to the accelerator. While the neural accelerator is still busy with handling the first batch, another input arrives. The CPU pre-processes the new input and makes it ready to be offloaded to the accelerator. The remaining neural computations of previous batch are handed over to CPU cores, and the accelerator simply moves on to the new input. The CPU takes over the unfinished neural layers of previous batch and then finalizes the inference. By pipelining the executions of disjoint neural network parts, NeuroPipe can effectively reduce the idle cycles of heterogeneous PUs and increase computational throughput. Importantly, it has an effect of relaxing a realtime constraint of neural processing and thus can handle faster inference in the resource-limited

TABLE 4. Workload characteristics of deep neural networks used in experiments.

Network	Operation count (GFLOPS)	Memory usage (MB)	Model parameters		Network description
			α	γ	
MLP [2]	0.02	121.25	0.32	11.0	MLP has a primitive form of network that stacks multiple fully-connected layers. It is still one of the most important and popularly-used neural networks.
ResNet [8]	7.2	246.76	0.18	18.7	ResNet implements a bottlneck-shaped structure and exploits 1×1 filters to reduce the amount of convolution operations.
MobileNet [25]	0.61	161.53	0.29	17.9	MobileNet divides convolutional layers into depth-wise and point-wise convolutions to reduce the total amount of operations.
YOLO [23]	44.9	771.25	0.07	37.4	YOLO reduces runtime by combining localization and classification, whereas other object detection networks such as (F)RCNN [5], [24] run them in series.
FCN [15]	11.1	463.50	0.13	36.8	FCN is comprised of only convolutional layers to retain the locational information of input data that are essential for semantic segmentation.

embedded edge device. These advantages open up hardware management opportunities in the edge device to drive energy saving or extra performance boosting by adjusting the operating voltage and frequency of embedded processor. As a result, NeuroPipe can significantly enhance the energy efficiency of neural processing in the edge device.

B. PERFORMANCE SPEEDUP MODEL

Performance speedup of NeuroPipe can be theoretically expressed as a function of i) relative duration of host operations in an inference interval of the baseline execution denoted as α and ii) relative performance ratio between the accelerator and CPU expressed as γ indicating the accelerator is γ times faster than the CPU for neural computations. For the baseline execution (see Figure 1a), suppose that the inference interval of a batch input is normalized to 1. Since host operations in the interval take α, the accelerator spends $1-\alpha$ for neural computations. When it comes to a pipelined execution based on NeuroPipe shown in Figure 1b, the duration of neural computations that can split into the CPU and accelerator becomes $1-2\alpha$ because the overlapped host operation time α has to be taken out from the total neural computation time of $1-\alpha$. Finding the right balance to partition the neural computations of $1-2\alpha$ between the CPU and accelerator yields $(1-2\alpha)\times\frac{\gamma}{\gamma+1}$. Consequently, the period of a pipeline phase becomes $\alpha+\frac{(1-2\alpha)\gamma}{\gamma+1}$, and the performance speedup of NeuroPipe over the baseline execution is expressed as Eq. (1) after rearranging the inverse of pipeline period.

$$\text{Speedup} = \frac{\gamma+1}{1-(\alpha-1)(\gamma-1)} \tag{1}$$

Table 4 lists DNN workloads used in our experiments and their computational traits including the α and γ parameters of performance speedup model, measured from NVIDIA Jetson AGX Xavier [4]. Among the list, MLP has a relatively larger α but smaller γ than other workloads, and YOLO shows the opposite characteristics. Other DNNs exhibit intermediate behaviors. In general, it is observed that lighter neural networks such as MLP and MobileNet tend to have larger α values since they spend relatively shorter duration of runtime on neural computations. Compute-bound workloads with intensive convolution operations such as YOLO and FCN manifest greater γ, which indicates that the accelerator better handles compute-intensive neural computations than the general-purpose CPU. If neural network workloads are more memory-bound or irregular, the performance gap dwindles between the neural accelerator and CPU as observed from other DNNs exhibiting smaller γ.

Figure 3 plots the theoretical performance speedup of NeuroPipe based on Eq. (1) for $\gamma = 10$ and 100 and α varied between 0 and 0.3. The actual speedup of five DNN workloads measured from NVIDIA Jetson AGX Xaiver using eight ARM CPU cores and 64 tensor cores are also marked in the graph. Plotting the theoretical model reveals several interesting insights into the efficacy of NeuroPipe. It effectively hides host CPU operations via pipelining, whereas the baseline execution runs them in series as shown in Figure 1. The pipelining effect is directly translated into a performance gain, and thus the DNNs with larger α exhibit greater performance speedup since more sequential host operations become overlapped and hidden. As a result, it makes an effect of eliminating the idle cycles of neural accelerator during the host operations.

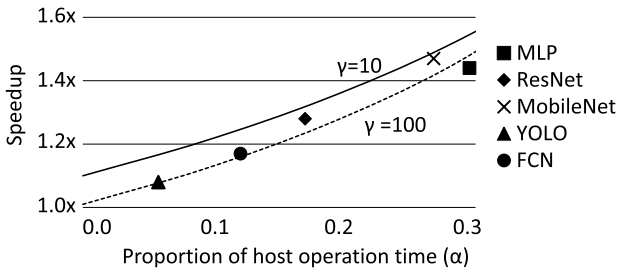


FIGURE 3. Theoretical performance speedup of NeuroPipe based on Eq. (1) as a function of host operation time (α) and performance ratio between a neural accelerator and CPU (γ). The actual performance speedup of five DNNs measured from NVIDIA Jetson AGX Xavier [4] are marked in the plot.

Engaging NeuroPipe with DNN executions in the edge device benefits more when there is a smaller performance gap (γ) between the neural accelerator and CPU. A small γ value implies that the neural accelerator can pass more neural layers to CPU cores and pipeline their executions, which has an effect of reducing the idle cycles of CPU cores.

Thus, neural networks exhibiting smaller γ generally show greater performance increases.

The performance speedup model can be extended to estimate the effect of multi-stage pipelining, while the proposed NeuroPipe technique takes a two-stage pipelining approach. If a neural network is divided into additional ω stages using the same set of heterogeneous PUs, the computational throughput of PUs per pipeline stage scales down by $1/\omega$ if a linear approximation is assumed. Calculating the performance speedup of multi-stage pipelining, it eventually leads to the identical equation in Eq. (1). The theoretical model indicates that two-stage pipelining is sufficient, and extending to multi-stage pipelining does not help eliminate more idle cycles of PUs since they have been already eradicated by two-stage pipelining. In fact, the computational throughput of PUs does not scale down by $1/\omega$ in that there is a diminishing return for increasing the number of PUs or threads [13]. Multi-stage pipelining can possibly exploit such an observation to extract additional performance via fine-grained DNN partitioning and pipelining. However, the attainable performance improvements are limited in a resource-limited edge device because DNN workloads crave for more compute resources than being saturated. For example, three-stage pipelined execution of MobileNet increases performance only by 2.5% over two-stage pipelining in NVIDIA Jetson AGX Xavier. Employing multi-stage pipelining is an overkill for significantly increased complexity of DNN partitioning and scheduling decisions, and thus the proposed NeuroPipe technique leverages two-stage pipelining.

C. DNN WORKLOAD PARTITIONING AND PIPELINING

A NeuroPipe scheduler figures out how to partition a neural network for heterogeneous PUs in an embedded edge device based on the theoretical model presented in Section III-B. In an initialization phase, the scheduler analyzes the computational characteristics of a given DNN workload by testing a few batch inputs using the conventional host-device execution shown in Figure 1a. In particular, it tries to measure i) the duration of host operations in an inference interval (α), ii) average performance difference between the accelerator and CPU for the whole neural network (γ), and iii) operation count per neural layer. The α and γ parameters are empirically collected from the hardware by measuring i) host operation time (t_{host}), ii) neural computation time when executed by the accelerator (t_{acc}) or CPU (t_{cpu}). The α and γ values are then calculated as $\alpha = t_{\text{host}}/(t_{\text{host}} + t_{\text{acc}})$ and $\gamma = t_{\text{cpu}}/t_{\text{acc}}$, respectively. The actual α and γ parameters of DNN workloads measured from NVIDIA Jetson AGX Xavier are shown in Table 4. In our experiments, testing only around four batch inputs gave us reliable estimates since neural computations are comprised of highly regular operations (e.g., matrix and/or vector multiplication and addition). Number of operations per neural layer are calculated offline from the composition of each neural layer instead of utilizing runtime profiling. For instance, the operation count of a fully-connected layer can be calculated simply as `input_size × output_size`, and that of a convolutional layer is `input_size × kernel_size × num_kernels`.

The measured metrics (i.e., α, γ, and operation count per layer) are used for determining a partitioning point to pipeline a chain of DNN inference with streaming inputs. The theoretical model described in Section III-B tells us that the total amount of neural operations per batch input is $(1-\alpha)\gamma$, and what can be passed from the accelerator to CPU cores is $N_{\text{opt}} = \frac{(1-2\alpha)\gamma}{\gamma+1}$ for optimal partitioning. Since NeuroPipe splits a neural network at layer granularity, an actual partitioning point is placed such that the total number of operations assigned to CPU cores become less than or equal to N_{opt}. Otherwise, passing more neural layers to CPU cores makes them excessively overloaded and results in longer inference intervals. DNNs in general have cone-shaped structures with diminishing layer sizes as the networks go deeper. Hence, the NeuroPipe scheduler assigns a set of consecutive layers at the rear of a neural network to CPU cores for better balancing of pipelined phases. In other words, NeuroPipe attempts to move many small layers as possible from the accelerator to CPU rather than relocating few large neural layers. Once a proper partitioning point is found in the given DNN workload, the NeuroPipe scheduler executes the split parts of DNN in pipelined manner following the execution sequence shown in Figure 1b.

Determining a partitioning point to pipeline the DNN inference based on three metrics (i.e., α, γ, and per-layer operation count) gives a very simple yet highly accurate solution as demonstrated in Table 5. It misses at most one neural layer in case of very deep DNNs such as ResNet and YOLO when comparing the outcomes of automated solution to empirically-found optimal results. The miscalculation causes performance degradation only by 0.2% on average, and thus exploiting more sophisticated partitioning solutions will not help much.

TABLE 5. Evaluation of DNN partitioning of NeuroPipe.

Neural network	Neural layer composition	Partitioning method	Assigned layers	
			Acc.	CPU
MLP [2]	6 fully-conn., 1 softmax	NeuroPipe	5	2
		Empirical	5	2
ResNet [8]	49 conv., 1 fully-conn., 1 softmax, and 18 others	NeuroPipe	64	5
		Empirical	65	4
MobileNet [25]	27 conv., 1 fully-conn., 1 softmax, and 1 pooling	NeuroPipe	27	3
		Empirical	27	3
YOLO [23]	75 conv. and 31 others	NeuroPipe	101	5
		Empirical	102	4
FCN [15]	8 conv. and 4 others	NeuroPipe	9	3
		Empirical	9	3

D. NEUROPIPE SCHEDULING DECISIONS

The pipelined execution of DNN inference delivers greater throughput over the baseline via cooperative executions between a neural accelerator and CPU in an embedded edge device. Eliminating the idle cycles of heterogeneous PUs also

helps NeuroPipe reduce energy consumption while achieving performance improvements by minimizing unnecessary static power dissipation during the idle periods. These advantages create hardware management opportunities to drive additional energy saving or performance boosting by adjusting the operating voltage and frequency of embedded processor. NeuroPipe makes such a scheduling decision based on a control flow shown in Figure 4. It analyzes the computational characteristics of a given DNN workload and finds an optimal partitioning point to pipeline the DNN inference based on the methodology described in Section III-C.

After partitioning the DNN, it is tested if the NeuroPipe execution without voltage and frequency adjustments satisfies a realtime constraint. If the timing constraint is met, NeuroPipe attempts to lower the operating voltage to conserve power while satisfying the realtime constraint. We refer to such an execution mode as *power-saving mode* (denoted as NP-P mode in Figure 4). Importantly, the power saving mode of NeuroPipe can substantially reduce power and energy consumption than the baseline execution for the same performance. In the opposite case that the simple engagement of NeuroPipe execution still fails to meet the realtime constraint of neural processing, it boosts up the execution for an additional performance increase while consuming about the same or even less energy than the baseline execution. This type of acceleration is referred to as *performance-boosting mode*. The definition of realtime constraint may differ depending on the requirements of embedded systems or applications. Instead of defining an arbitrary realtime constraint, we suppose the timing of baseline execution as a target constraint in this paper to demonstrate how various NeuroPipe execution modes compare to the baseline scheme. The remainder of the paper is presented based on this assumption. In the following section, we evaluate the performance, power, energy, and thermal implications of various NeuroPipe execution schemes based on hardware measurements.

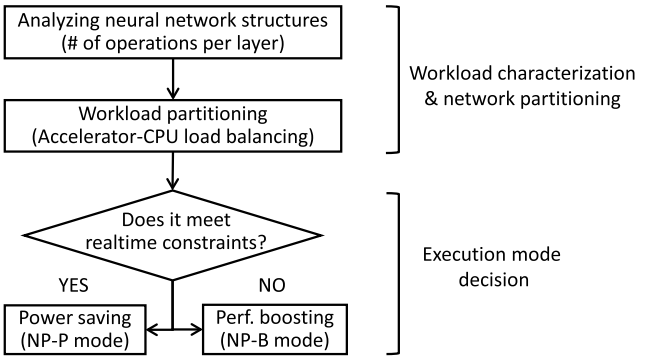


FIGURE 4. Decision flow of neural network partitioning for co-acceleration of heterogeneous processing units. The scheduler determines how the neural networks are split to the PUs after analyzing the network structure and relative performance ratio between accelerator and CPU. Then, it selects a proper execution scheme between power-saving and boosting modes depending on the realtime constraint.

IV. EVALUATION

A. EXPERIMENT ENVIRONMENT

We used NVIDIA Jetson AGX Xavier [4] in our experiments as an exemplary embedded edge device that integrates eight ARM CPU cores and 64 tensor cores as neural accelerators in the same processor package. The performance, power, energy, and thermal implications of NeuroPipe are analyzed based on hardware measurements and compared against the baseline execution as a target constraint that performs a simple host-device execution scheme. The edge device used in our experiments carries a vendor-provided software tool named *tegrastats* [3] that enables us to measure power, clock frequency, voltage, and temperature of discrete processing units in the embedded processor. We modified system files in the built-in operating system to adjust the operating voltage and frequency of processing units to apply the NeuroPipe execution schemes (i.e., power-saving and performance-boosting modes) based on the decision flow described in Figure 4. We implemented a lightweight C++ framework [13] tailored for the edge device by adopting various features from the well-known Caffe framework [12] but discarding unnecessary software elements to make the neural network framework as light as possible. Hence, the native computational traits of neural networks are better manifested in hardware measurement results via the deployment of lightweight framework on the edge device. Various NeuroPipe execution modes are assessed with five representative DNN workloads including MLP [2], ResNet [8], MobileNet [25], YOLO [23], and FCN [15] as summarized in Table 4.

B. BASIC MODE OF NEUROPIPE

The proposed NeuroPipe technique partitions a neural network into groups of consecutive layers as drawn in Figure 1b and pipelines their executions using different types of heterogeneous PUs in an embedded edge device. By overlapping the inference of streaming inputs via cooperative executions between the neural accelerator and CPU, NeuroPipe minimizes the idle periods of processing units and thus achieves greater throughput over the baseline scheme (i.e., conventional host-device execution) in the resource-limited embedded edge device.

Figure 5 shows the execution time, power, energy, and energy-delay product (EDP) of NeuroPipe with the DNN workloads listed in Table 4. In this experiment, NeuroPipe partitions and pipelines the DNNs, but it does not adjust the operating voltage and frequency of processing units. We refer to such a simplest execution scheme of NeuroPipe as *basic mode* to distinguish it from other operation modes such as power-saving and performance-boosting modes. Measurement results show that the basic mode of NeuroPipe reduces the execution time of DNNs by 17.6% on average and as much as 32.0% in case of MobileNet as plotted in Figure 5a. Speedup differences between the DNNs can be easily explained by referring to the theoretical performance speedup model in Eq. (1) and its graph plotted in Figure 3.

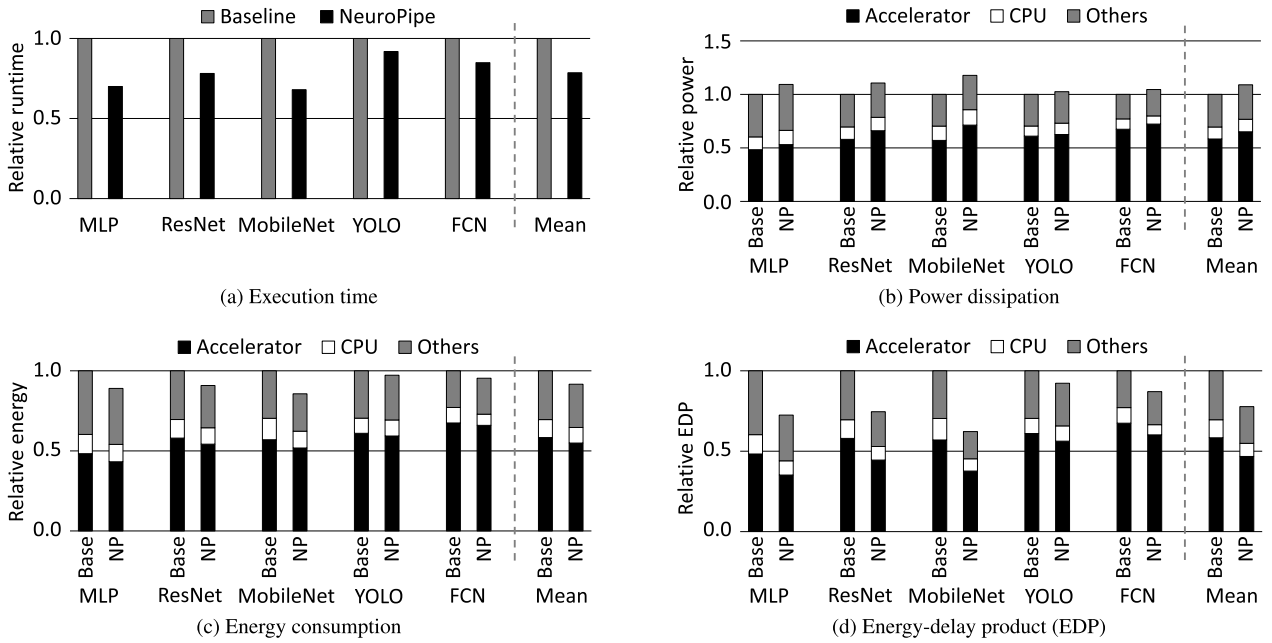


FIGURE 5. Evaluation of basic-mode NeuroPipe executions (denoted as NP in the graphs) without voltage and frequency adjustments compared with the baseline (shown as Base) for (a) execution time, (b) power, (c) energy, and (d) EDP.

The performance improvements come i) primarily from the overlapped executions of host CPU and neural accelerator and ii) partly from sharing neural computations between the heterogeneous PUs. As illustrated in Figure 1, NeuroPipe effectively hides host CPU operations via pipelining, whereas the conventional host-device execution serializes them. Thus, small networks such as MLP and MobileNet exhibiting large α (i.e., relative duration of host operations in a batch inference interval) as specified in Table 4 tend to show greater performance enhancements because the neural accelerator becomes active during the host operation time instead of wasting idle cycles. Sharing neural computations between the heterogeneous PUs additionally contributes to performance increases. Since there is a large throughput gap in general between the neural accelerator and CPU, it limits the number of neural layers that can be passed from the accelerator to CPU cores to parallelize the inference in pipelined manner. Sharing the neural computations between the heterogeneous PUs is especially beneficial when there is a smaller throughput difference between the accelerator and CPU (i.e., smaller γ) such as ResNet and MobileNet. Neural networks exhibiting smaller γ values imply that they have relatively more computational irregularity than other DNNs, and therefore CPU cores handle them comparably well. As a result, it makes an effect of eliminating the idle cycles of CPU cores by sharing the neural computations with the accelerator.

In addition to performance enhancements, Figure 5c reveals that the basic mode of NeuroPipe also reduces energy consumption by 8.4% on average and as much as 14.4% in case of MobileNet. The energy saving originates from removing the idle cycles of neural accelerator and CPU cores by pipelining their executions. Idle processing units in the baseline execution scheme may be power-gated to conserve unnecessary power dissipation, but sub-second inference intervals make it less practical to apply fine-grained power controls to frequently and repeatedly shut off and on. Consequently, Figure 5d shows that the basic mode of NeuroPipe diminishes the EDP by 22.3% on average.

A downside of NeuroPipe lies in increased power dissipation as shown in Figure 5b. The graph plots the average of periodically sampled data collected from power meters in the edge device during the runtime execution of each neural network. Transient power behaviors of NeuroPipe and baseline executions are shown in Figure 6 with MobileNet as an example. The graphs disclose that the baseline execution exhibits highly fluctuating power traces for both accelerator and CPU, and the total power has a peak-to-peak variation of 3.24W. On the other hand, NeuroPipe shows nearly constant and thus stable power dissipation for both types of heterogeneous PUs. The peak-to-peak variation of total power in this case is only 0.31W. Figure 5b plotting the average of varying power numbers tells us that the basic mode of NeuroPipe consumes 8.9% greater power than the baseline execution. However, the runtime power traces in Figure 6 reveal that the max power dissipation of NeuroPipe is in fact only 3.3% higher than that of the baseline. Thus, the power penalty is not as significant as what Figure 5b may delusively show.

C. POWER-SAVING MODE

In case the baseline execution of a DNN by itself satisfies a realtime constraint in an embedded edge device, NeuroPipe can translate performance improvements obtained by

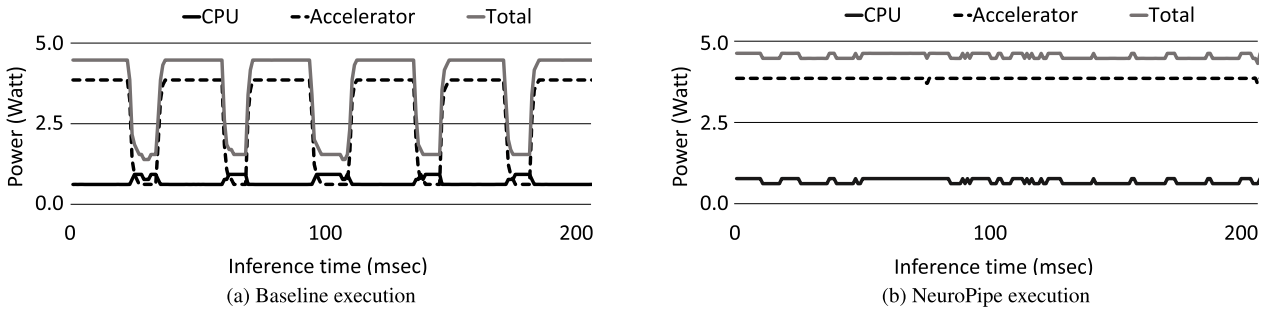


FIGURE 6. Runtime power behaviors of (a) baseline and (b) NeuroPipe executions for MobileNet. NeuroPipe exhibits stable power consumption, whereas the baseline shows highly fluctuating traces at the inference boundaries of discrete batch inputs.

pipelining into power saving based on the decision flow shown in Figure 4. The power-saving mode of NeuroPipe utilizes the voltage and frequency scaling capability of embedded processor. In particular, it decreases the operating voltage and frequency of processing units to a level that produces nearly the same performance as the baseline execution such that the realtime constraint can still be met.

Figure 7 plots the resulting execution time and average power of NeuroPipe with power-saving mode. Figure 7a proves that the power-saving mode has calibrated clock frequency to yield about the same execution time as the baseline execution. In the meantime, reducing the operating voltage saves the power by 11.4% on average as shown in Figure 7b. In other words, the power-saving mode of NeuroPipe delivers the same performance as the baseline at lower power. Energy and EDP graphs are not included in Figure 7 because equalizing the execution time between the baseline and power-saving mode of NeuroPipe produces energy and EDP graphs nearly identical to the power plot shown in Figure 7b.

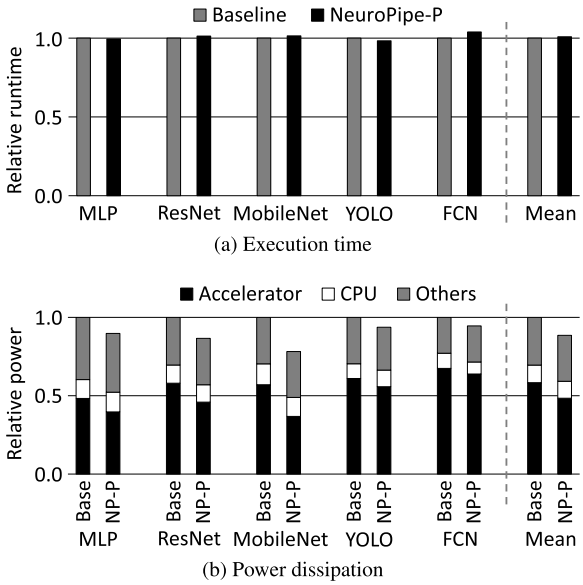


FIGURE 7. (a) Execution time and (b) power dissipation of NeuroPipe with power-saving mode (NP-P) compared with the baseline execution (Base). The power-saving mode turns the extra performance into power reduction by decreasing the operating voltage and frequency, thereby achieving low-power executions for the same performance.

D. PERFORMANCE-BOOSTING MODE

As an opposite scheme of power-saving mode, NeuroPipe with performance-boosting mode trades in the energy saving obtained by pipelining for an additional performance gain in case a greater throughput is demanded to meet a realtime constraint in an embedded edge device. The decision is also made based on the management flow described in Figure 4. Similar to the power-saving mode, the performance-boosting mode also utilizes the voltage and frequency scaling capability of embedded processor to adjust the operating conditions.

Figure 8 demonstrates how much extra performance can be achieved by trading the energy saving of basic-mode NeuroPipe, and the resulting power and EDP are shown. Figure 8a shows that NeuroPipe with performance-boosting mode reduces the execution time by 30.5% on average compared to the baseline, and it gains 12.9% performance increase over the basic mode of NeuroPipe (i.e., NeuroPipe without voltage and frequency adjustments). As plotted in Figure 8c, the performance-boosting mode of NeuroPipe transforms the energy savings into extra performance enhancements under the same energy budget as the baseline execution. In other words, NeuroPipe with performance-boosting mode achieves far more performance increases over the baseline for the same energy consumption. Of course, the performance-boosting mode can further drive performance improvements beyond the par-energy point, but it will not make a fair comparison with the baseline and thus is not discussed here. Consequently, Figure 8d reveals that NeuroPipe with performance-boosting mode reduces the EDP by 32.8% on average. The primary drawback of performance-boosting mode is the increased power dissipation as shown in Figure 8b. The graph shows that it results in 50.8% higher power than the baseline execution for the 30.5% average performance increase. There apparently exists a diminishing performance return for the power increase.

E. THERMAL IMPLICATIONS

The proposed NeuroPipe technique accomplishes performance improvements at lower energy by concurrently engaging heterogeneous PUs in neural computations. Except for the power-saving mode of NeuroPipe, the concurrent executions of neural accelerator and CPU result in increased

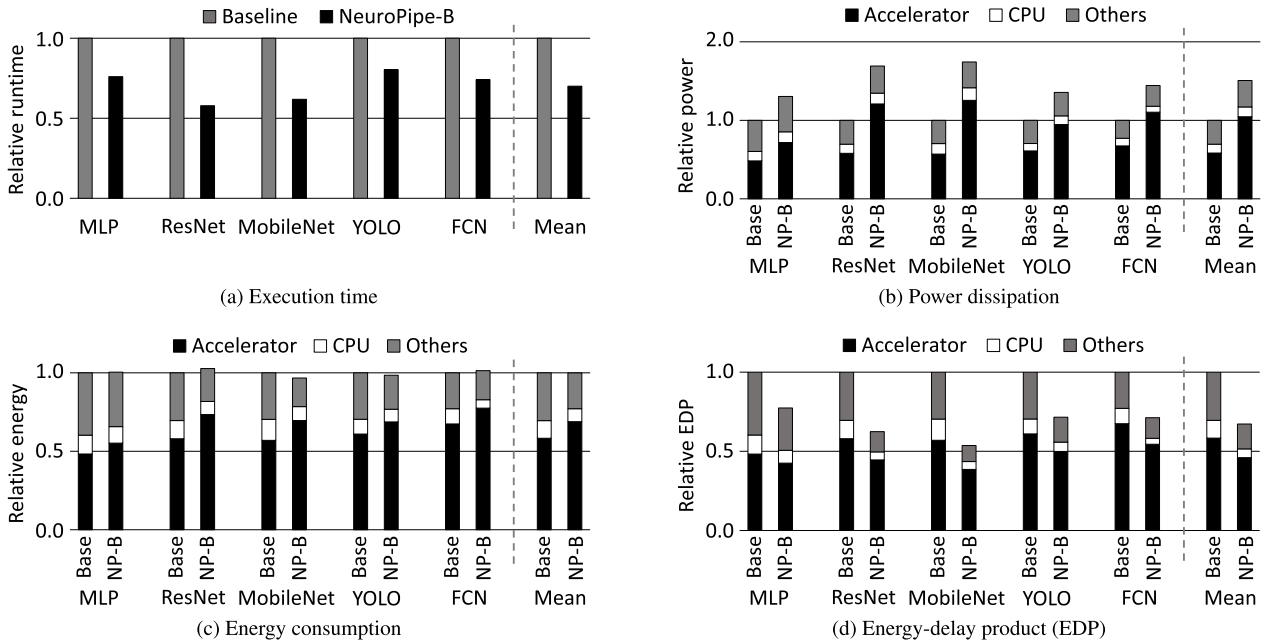


FIGURE 8. (a) Execution time, (b) power, (c) energy, and (d) EDP of NeuroPipe with performance-boosting mode (NP-B) compared with the baseline execution (Base). In the performance-boosting mode, NeuroPipe trades in energy saving for an additional performance gain by increasing the operating voltage and frequency.

power dissipation that is unfavorable in a thermal aspect. As plotted in Figure 6, NeuroPipe exhibits constant power traces unlike the baseline execution that has highly fluctuant power behaviors. The stable power drawing of NeuroPipe may be favorable in a power-delivery aspect, but it may raise a concern for increased thermal coupling between the active PUs since NeuroPipe fully utilizes the heterogeneous PUs without idle periods.

Figure 9 compares the measured steady-state temperatures of various NeuroPipe execution modes and the baseline with two different cooling mechanisms. We notice that there already exists a strong thermal coupling between the CPU and accelerator in the baseline execution in that the CPU temperature regardless of its activity is dominated by that of the accelerator. Since there is little temperature difference between the CPU and accelerator, Figure 9 simply plots a single temperature value for each execution scheme because the measured temperature differences were within the precision of thermal sensors (± 1°C). Although the basic mode of NeuroPipe increases power dissipation by concurrently activating all heterogeneous PUs, it results in only 2.5°C increase at most compared to the baseline execution. However, the performance-boosting mode of NeuroPipe causes up to 19.0°C temperature increase in case of FCN for the additional performance gain when passive cooling is used. Figure 9b shows that the thermal problem of NeuroPipe with performance-boosting mode can be suppressed by employing an active cooling mechanism, and the use of forced-air fan decreases the temperature gap (between the baseline and performance-boosting mode) down to 8.0°C in the worst case. Thus, experiment results demonstrate that NeuroPipe does not incur serious thermal problems compared

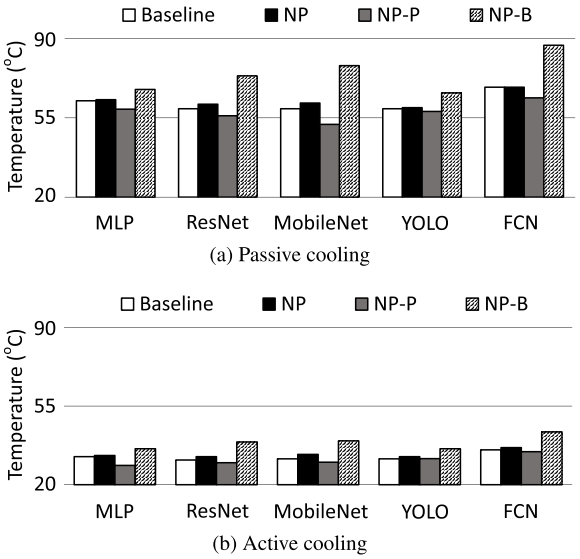


FIGURE 9. Comparison of steady-state temperatures of various NeuroPipe execution modes and the baseline scheme for (a) passive cooling only with a heatsink and (b) active cooling with a forced-air fan.

to the baseline, and in case of performance-boosting mode employing active cooling can amortize the thermal cost.

V. CONCLUSION

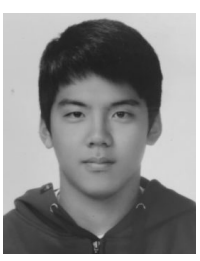
Deep learning on edge devices has potentially many advantages such as security and latency, but there are on-going challenges to accelerate neural computations in realtime-constrained embedded edge devices in energy-efficient manner. This paper presented a novel hardware management technique named *NeuroPipe* that enables cooperative

acceleration of heterogeneous PUs in the embedded edge devices for faster DNN inference at lower energy. The following summarizes the key contributions of NeuroPipe.

- NeuroPipe proposes a novel hardware management solution that achieves performance enhancements at lower energy for neural processing in a resource-limited embedded edge device incorporating a neural accelerator.
- The power-saving mode of NeuroPipe substantially decreases power and energy consumption of DNN inference for the same performance as the conventional execution scheme.
- The performance-boosting mode translates the energy saving of NeuroPipe into an extra performance gain under the same energy budget as the baseline.
- Hardware measurement results of NeuroPipe demonstrate that its various execution schemes significantly enhance the energy efficiency of neural processing in the embedded edge device.

REFERENCES

[1] S. Cass, "Taking AI to the edge: Google's TPU now comes in a maker-friendly package," *IEEE Spectr.*, vol. 56, no. 5, pp. 16–17, May 2019.
[2] D. C. Cireşan, U. Meier, L. M. Gambardella, and J. Schmidhuber, "Deep, big, simple neural nets for handwritten digit recognition," *Neural Comput.*, vol. 22, no. 12, pp. 3207–3220, Dec. 2010.
[3] NVIDIA Corporation. (Dec. 2019). *NVIDIA Jetson Linux Driver Package Developer Guide, 32.3.1 Release*. [Online]. Available: https://docs.nvidia.com/jetson/l4t/
[4] M. Ditty, A. Karandikar, and D. Reed, "Nvidia's xavier SoC," in *Proc. IEEE Hot Chips Symp. (HCS)*, Aug. 2018, pp. 1–17.
[5] R. Girshick, J. Donahue, T. Darrell, and J. Malik, "Rich feature hierarchies for accurate object detection and semantic segmentation," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, Jun. 2014, pp. 580–587.
[6] S. Han, H. Mao, and W. Dally, "Deep compression: Compressing deep neural networks with pruning, trained quantization and Huffman coding," in *Proc. Int. Conf. Learn. Represent.*, May 2016, pp. 1–14.
[7] S. Han, J. Pool, J. Tran, and W. J. Dally, "Learning both weights and connections for efficient neural networks," in *Proc. Int. Conf. Neural Inf. Process. Syst.*, Dec. 2015, pp. 1135–1143.
[8] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit. (CVPR)*, Jun. 2016, pp. 770–778.
[9] A. Howard, M. Sandler, B. Chen, W. Wang, L.-C. Chen, M. Tan, G. Chu, V. Vasudevan, Y. Zhu, R. Pang, H. Adam, and Q. Le, "Searching for MobileNetV3," in *Proc. IEEE/CVF Int. Conf. Comput. Vis. (ICCV)*, Oct. 2019, pp. 1314–1324.
[10] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam, "MobileNets: Efficient convolutional neural networks for mobile vision applications," 2017, *arXiv:1704.04861*. [Online]. Available: http://arxiv.org/abs/1704.04861
[11] B. Jacob, S. Kligys, B. Chen, M. Zhu, M. Tang, A. Howard, H. Adam, and D. Kalenichenko, "Quantization and training of neural networks for efficient integer-arithmetic-only inference," in *Proc. IEEE/CVF Conf. Comput. Vis. Pattern Recognit.*, Jun. 2018, pp. 2704–2713.
[12] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell, "Caffe: Convolutional architecture for fast feature embedding," in *Proc. ACM Int. Conf. Multimedia (MM)*, 2014, pp. 675–678.
[13] B. Kim, S. Lee, C. Park, H. Kim, and W. Song, "The nebula benchmark suite: Implications of lightweight neural networks," *IEEE Trans. Comput.*, early access, Oct. 7, 2020, doi: 10.1109/TC.2020.3029327.
[14] J. Lin, Y. Rao, J. Lu, and J. Zhou, "Runtime neural pruning," in *Proc. Int. Conf. Neural Inf. Process. Syst.*, Dec. 2017, pp. 2181–2191.
[15] J. Long, E. Shelhamer, and T. Darrell, "Fully convolutional networks for semantic segmentation," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit. (CVPR)*, Jun. 2015, pp. 3431–3440.
[16] P. Micikevicius, S. Narang, J. Alben, G. Diamos, E. Elsen, D. Garcia, B. Ginsburg, M. Houston, O. Kuchaiev, G. Venkatesh, and H. Wu, "Mixed precision training," in *Proc. Int. Conf. Learn. Represent.*, May 2018, pp. 1–12.
[17] S. Mittal and J. S. Vetter, "A survey of CPU-GPU heterogeneous computing techniques," *ACM Comput. Surv.*, vol. 47, no. 4, pp. 1–35, Jul. 2015.
[18] D. Moloney, B. Barry, R. Richmond, F. Connor, C. Brick, and D. Donohoe, "Myriad 2: Eye of the computational vision storm," in *Proc. IEEE Hot Chips 26 Symp. (HCS)*, Aug. 2014, pp. 1–18.
[19] D. Narayanan, A. Harlap, A. Phanishayee, V. Seshadri, N. R. Devanur, G. R. Ganger, P. B. Gibbons, and M. Zaharia, "PipeDream: Generalized pipeline parallelism for DNN training," in *Proc. 27th ACM Symp. Operating Syst. Princ.*, Oct. 2019, pp. 1–15.
[20] S. Ohshima, K. Kise, T. Katagiri, and T. Yuba, "Parallel processing of matrix multiplication in a CPU and GPU heterogeneous environment," in *Proc. Int. Conf. High Perform. Comput. Comput. Sci.*, Jun. 2006, pp. 305–318.
[21] Qualcomm Technologies. (Sep. 2020). *Snapdragon Neural Processing Engine SDK: Reference Guide*. 80-NL315-14. [Online]. Available: https://developer.qualcomm.com/docs/snpe/index.html
[22] M. Quartermain, T. Arora, and R. Jagtap. (Mar. 2020). *Technical Overview of the ARM Cortex-M55 and Ethos-U55 Processors*. [Online]. Available: https://www.brighttalk.com/webcast/17792/391867.
[23] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi, "You only look once: Unified, real-time object detection," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit. (CVPR)*, Jun. 2016, pp. 779–788.
[24] S. Ren, K. He, R. Girshick, and J. Sun, "Faster R-CNN: Toward real-time object detection with region proposal networks," in *Proc. Int. Conf. Neural Inf. Process. Syst.*, Dec. 2015, pp. 91–99.
[25] 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.
[26] H. Sharma, J. Park, N. Suda, L. Lai, B. Chau, V. Chandra, and H. Esmaeilzadeh, "Bit fusion: Bit-level dynamically composable architecture for accelerating deep neural network," in *Proc. ACM/IEEE 45th Annu. Int. Symp. Comput. Archit. (ISCA)*, Jun. 2018, pp. 764–775.
[27] E. Talpes, D. D. Sarma, G. Venkataramanan, P. Bannon, B. McGee, B. Floering, A. Jalote, C. Hsiong, S. Arora, A. Gorti, and G. S. Sachdev, "Compute solution for Tesla's full self-driving computer," *IEEE Micro*, vol. 40, no. 2, pp. 25–35, Mar. 2020.
[28] 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 Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 39, no. 10, pp. 2254–2267, Oct. 2020.
[29] J. Yang, X. Shen, J. Xing, X. Tian, H. Li, B. Deng, J. Huang, and X. Hua, "Quantization networks," in *Proc. IEEE/CVF Conf. Comput. Vis. Pattern Recognit.*, Jun. 2019, pp. 7300–7308.
[30] M. Yang, S. Wang, J. Bakita, T. Vu, F. D. Smith, J. H. Anderson, and J.-M. Frahm, "Re-thinking CNN frameworks for time-sensitive autonomous-driving applications: Addressing an industrial challenge," in *Proc. IEEE Real-Time Embedded Technol. Appl. Symp. (RTAS)*, Apr. 2019, pp. 305–371.
[31] X. Zhang, X. Zhou, M. Lin, and J. Sun, "ShuffleNet: An extremely efficient convolutional neural network for mobile devices," in *Proc. IEEE/CVF Conf. Comput. Vis. Pattern Recognit.*, Jun. 2018, pp. 6848–6856.
[32] Z. Zhong, V. Rychkov, and A. Lastovetsky, "Data partitioning on multicore and multi-GPU platforms using functional performance models," *IEEE Trans. Comput.*, vol. 64, no. 9, pp. 2506–2518, Sep. 2015.

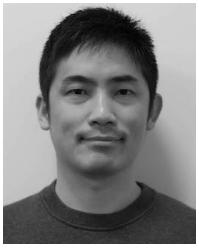


BOGIL KIM received the B.S. degree in electrical engineering from Yonsei University, Seoul, South Korea, in 2018, where he is currently pursuing the Ph.D. degree with the School of Electrical and Electronic Engineering. His research interests and studies are geared towards workload characterization and optimization of neural networks, hardware accelerators, mobile computing, and memory systems.

SUNGJAE LEE received the B.S. degree in electrical engineering from Yonsei University, Seoul, South Korea, in 2019, where he is currently pursuing the degree with the School of Electrical and Electronic Engineering.

His research interests include high-performance memory systems and architectures, workload characterization of emerging applications, and microarchitecture modeling and simulation techniques.

AMIT RANJAN TRIVEDI (Member, IEEE) received the B.Tech. and M.Tech. degrees in electrical engineering from IIT Kanpur, Kanpur, India, in 2008, and the Ph.D. degree in electrical and computer engineering from the Georgia Institute of Technology, Atlanta, GA, USA, in 2015. He was with the IBM Semiconductor Research and Development Center (SRDC), Bengaluru, India, from 2008 to 2010. During his Ph.D., he was a Summer Intern with the IBM T. J. Watson Research Institute, Yorktown Heights, NY, USA, in 2012, and the Circuits Research Laboratory, Intel, Hillsboro, OR, USA, in 2014. Since 2015, he has been with the Department of Electrical and Computer Engineering, University of Illinois at Chicago, Chicago, IL, USA, where he is currently an Assistant Professor. He has authored or coauthored over 40 articles in refereed journals and conferences. His current research interests include machine learning at the edge, hardware security, and energy-efficient systems. He received the IEEE Electron Device Society Fellowship in 2014, where he was one of the three recipients worldwide. He also received the Georgia Tech Sigma Xi Best Ph.D. Dissertation Award.



WILLIAM J. SONG (Member, IEEE) received the B.S. degree in electrical engineering from Yonsei University, Seoul, South Korea, and the Ph.D. degree in electrical and computer engineering from the Georgia Institute of Technology, Atlanta, GA, USA. He was a Senior Engineer with Intel, Santa Clara, CA, USA. He was a Graduate Research Intern with Qualcomm, San Diego, CA, USA, in 2015, the IBM T. J. Watson Research Center, Yorktown Heights, NY, USA, in 2014 summer and fall, the AMD Research, Bellevue, WA, USA, in 2013 summer, and the Sandia National Laboratories, Albuquerque, NM, USA, from 2010, 2011, and 2012 summers. He is currently an Assistant Professor with the School of Electrical Engineering, Yonsei University. His research focus lies in the challenges of heterogeneous architectures, processing near data for deep learning, big data applications, solutions to power, thermal, and reliability issues in many-core microarchitectures, and 3-D-integrated packages. He was a recipient of the IBM/SRC Graduate Fellowship from 2012 to 2015. He received the Best in Session Award from SRC TECHCON in 2014 and the Best Student Paper Award from the IEEE International Reliability Physics Symposium in 2015. He received the Distinguished Faculty Award from Yonsei University in 2018 and the Teaching Excellence Award in 2018 and 2020.