Received 19 February 2024, accepted 6 March 2024, date of publication 11 March 2024, date of current version 18 March 2024.

Digital Object Identifier 10.1109/ACCESS.2024.3375517

RESEARCH ARTICLE

A Novel Throughput Enhancement Method for Deep Learning Applications on Mobile Devices With Heterogeneous Processors

CHOONGHOON PARK AND SOONHOI HA, (Fellow, IEEE)
Department of Computer Science and Engineering, Seoul National University, Seoul 08826, South Korea

Corresponding author: Soonhoi Ha (sha@snu.ac.kr)

This work was supported in part by Samsung Electronics Company Ltd., under Grant IO201210-07941-01; and in part by the Institute of Computer Technology (ICT), Seoul National University.

ABSTRACT Contemporary smartphones integrate dedicated AI accelerators alongside CPUs and GPUs, in response to the growing demand for deep learning applications. While existing software development kits (SDKs) for these devices provide neural network optimization techniques, they often lack system-level optimizations, specifically in distributing layers across heterogeneous processors. This paper introduces a novel approach to enhance the throughput of deep learning applications through the utilization of quantization and pipelining techniques. The proposed technique employs different quantization schemes for activation data and filter weights to minimize accuracy drop. A genetic algorithm is employed to explore the extensive design space of layer-wise mapping and pipelining, aiming to find the best pipelining solution. To estimate performance of each solution candidate, the actual execution time of the application on the device is measured, accounting for unique smartphone characteristics, such as dynamic voltage and frequency scaling (DVFS) and OS scheduling. The impact of thermal throttling on throughput is also investigated by running benchmark applications continuously for 10 minutes. Our technique is validated through experiments conducted on Google Pixel 6 and Samsung Galaxy S22. Throughput enhancements, ranging from $\times$5.4 to $\times$7.6 on Google Pixel 6 and $\times$35.5 to $\times$44.2 on Samsung Galaxy S22, are achieved, compared to single-processor mappings for networks with floating-point parameters. It confirms that significant performance improvements can be achieved through the proposed software optimization methodology on contemporary smartphones with diverse constraints at the user level.

INDEX TERMS Mobile devices, heterogeneous processors, quantization, pipelining, design space exploration.

I. INTRODUCTION

As the demand for deep learning applications continues to rise, the emergence of embedded devices equipped with heterogeneous processors facilitates the efficient execution of deep learning(DL) tasks. Notable examples include advanced AI hardware platforms like the NVIDIA Jetson AGX Xavier board [1] and Google Coral Dev board [2], as well as modern smartphones like the Google Pixel 6 [3] and Samsung Galaxy S22 [4]. These devices incorporate dedicated AI accelerators alongside CPUs and GPUs. The accompanying software development kits (SDKs) for these devices include libraries designed for the efficient execution of AI applications. While these libraries offer optimization techniques for the neural network itself, such as quantization [5] and pruning [6], they fall short in providing system-level optimization techniques. There is a lack of methodologies to effectively distribute layers across various heterogeneous processors.

A methodology has been proposed to maximize the throughput of a single deep learning (DL) application using both GPU and NPU, leveraging various types of parallelism [7]. This approach is implemented on TensorRT, the

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

software development kit (SDK) for NVIDIA Jetson boards. It enhances throughput by partitioning the DL network into multiple stages and mapping each stage to different processors, enabling pipelined execution. The methodology incorporates multithreading to boost the utilization of each processing element by overlapping multiple executions of the network. Their proposed parallelization method reflects the limitations of the hardware platform and SDK: For instance, the CPU is exclusively utilized for pre-processing or post-processing tasks and is not involved in the network body.

In this paper, we propose an optimization technique aimed at enhancing the throughput of deep learning applications on commercial smartphones equipped with AI accelerators. Our focus is particularly on quantization and pipelining. Unlike the aforementioned work, our methodology allows each partitioned inference stage not only to be mapped to the GPU and accelerator but also to CPUs, enabling a more fine-grained inference pipelining. We account for the varying processing speeds of each processor after quantization. Importantly, our approach considers a unique characteristic of smartphones: the operating system implements DVFS (dynamic voltage and frequency scaling) or a specialized thermal throttling scheme to prevent the temperature from exceeding a certain range.

While previous research typically relies on profiling-based heuristics for parallelizing deep learning networks, our approach employs a genetic algorithm based on direct performance measurements for layer-wise mapping exploration, similar to [7]. We employ different quantization schemes for activation data and filter weights to minimize the accuracy drop caused by quantization. Since increasing processor utilization may elevate the device temperature, it is likely for smartphones to employ thermal throttling mechanisms to limit temperature rise. We also investigate the impact of thermal throttling on throughput performance by running benchmark applications continuously for 10 minutes.

The effectiveness of our proposed technique is demonstrated through validation with various deep learning networks on Google Pixel 6 and Samsung Galaxy S22. In comparison to single processor mappings for networks with floating-point parameters, our method significantly enhances the throughput of deep learning applications, ranging from $\times 5.4$ to $\times 7.6$ for each network on Google Pixel 6 and from $\times 35.5$ to $\times 44.2$ on Samsung Galaxy S22.

The main contributions of this study can be summarized as follows:

- We propose a novel throughput enhancement technique by leveraging both quantization and parallelization on commercial smartphones equipped with AI accelerators, CPUs, and GPUs.
- The performance evaluation is conducted through direct measurements on the device, ensuring that the impact of device-specific constraints imposed by each smartphone is naturally considered in the proposed mapping exploration method.
- We investigate how thermal throttling mechanisms may impact the throughput performance for each device.
- Our approach yields substantial throughput improvements on both Google Pixel 6 and Samsung Galaxy S22 smartphones; we observe a wide variance in the degree of improvement depending on the hardware characteristics.

The remaining sections of the paper are organized as follows. Section II provides a review of quantization and pipelining techniques laying the foundation for the methodology employed in our study. We present the hardware specification details of two commercial smartphones used in experiments: Galaxy S22 and Google Pixel 6, Section III provides a review of related works on deep learning optimization for smartphones with limited resources, Section IV explains the proposed technique for throughput enhancement. After presenting the experimental setup, results, and discussion in Section V, we conclude the paper, presenting future research topics in Section VI.

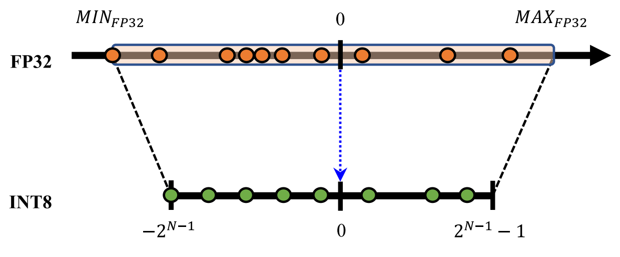
II. BACKGROUND

A. QUANTIZATION

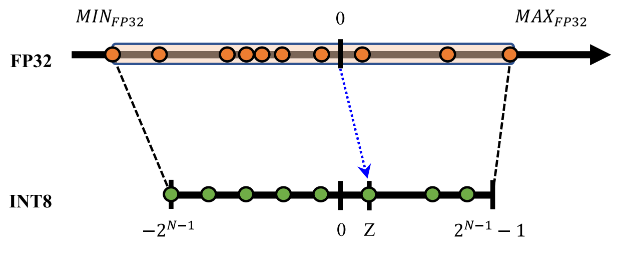
Accelerators incorporated into commercial mobile devices are typically fine-tuned for INT8 (8-bit integer) operations. To maximize the efficiency of these accelerators, the conversion of FP32 (32-bit floating-point) networks to INT8 networks through quantization becomes imperative. Quantization, being a highly effective technique, is extensively explored in deep learning optimization due to its ability to significantly decrease memory requirements and computation latency [8].

Due to challenges in efficiently deploying non-uniform quantization schemes on processing elements, uniform quantization has become the prevailing method. The uniform quantization process involves determining quantization parameters, scales and zero-points. Equations (1) to (5) illustrate the process of obtaining scales and zero-points for quantization. Equation (1) maps FP32 values to INT8 values using scales and zero-points, truncating the fractional part of FP32 with the round function. This function, which can employ rounding, truncation, or adaptive rounding [11], is predominantly used with adaptive rounding to reduce accuracy drop. Equation (2) represents the formula for calculating scales, where the numerator denotes the length of the FP32 range, and the denominator indicates the length of the INT8 range for scaling. The FP32 range, computed using minimum and maximum values as in Equations (3) and (4), varies based on whether the quantization is symmetric or asymmetric. Equation (5) calculates zero-points, determining the INT8 value corresponding to the FP32 0 value using the minimum value of the FP32 range and scales. In this paper, N is fixed at 8 since INT8 quantization is applied to both weights and activations.

$$X_{INT8} = \lceil \frac{X_{FP32}}{S} + Z \rfloor \tag{1}$$

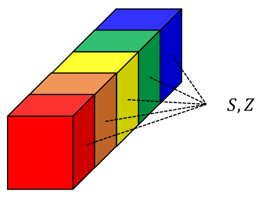


(a) Symmetric quantization

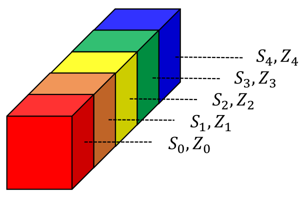


(b) Asymmetric quantization

FIGURE 1. Comparative illustration of symmetric and asymmetric quantization.



(a) Per-tensor quantization



(b) Per-channel quantization

FIGURE 2. Comparative illustration of per-tensor and per-channel quantization.

$$S = \frac{\beta - \alpha}{2^N - 1} \tag{2}$$

$$\alpha = \begin{cases} -max(|X_{FP32}|), & \text{symmetric} \\ min(X_{FP32}), & \text{asymmetric} \end{cases} \tag{3}$$

$$\beta = \begin{cases} max(|X_{FP32}|), & \text{symmetric} \\ max(X_{FP32}), & \text{asymmetric} \end{cases} \tag{4}$$

$$Z = -\lceil \frac{\alpha}{S} - 2^{N-1} \rfloor \tag{5}$$

Figure 1 illustrates the difference between symmetric and asymmetric quantization schemes. The choice between these schemes depends on the distribution of tensor values. Asymmetric quantization holds an advantage over symmetric quantization when the distribution is highly asymmetric. The determination of the range can be applied to the entire tensor or specific channels within it. In the per-channel quantization scheme, depicted in Figure 2, quantization parameters are determined differently for each channel.

Quantization methods can be broadly classified into two categories: Quantization-Aware Training (QAT) and Post-Training Quantization (PTQ). QAT involves retraining the weights of pre-trained FP32 networks, taking into account quantization considerations [9]. This approach often achieves higher accuracy than PTQ as the weights are optimized for quantization. It comes with the drawback of significant retraining overhead, and may not be feasible in the absence of training data. On the other hand, PTQ uses pre-trained FP32 networks as a starting point but performs quantization to INT8 parameters without retraining, relying on a limited data set for calibration [10]. While PTQ allows for a quick conversion to an INT8 network, it tends to result in lower accuracy than QAT. To address this limitation, various techniques have been developed, including AdaRound(adaptive rounding) [11], and cross-layer equalization and bias correction [12]. Cross-layer equalization leverages the scale-equivariance property of activation functions to equalize the weight range within the network for more effective use of representative bits. Bias correction corrects biases in layer output values introduced by quantization. Thanks to these methods, PTQ is recognized for its ability to achieve 8-bit quantization with accuracy close to floating-point precision in most cases [13].

TABLE 1. Specifications of mobile devices used in this work.

Device	Google Pixel 6	Samsung Galaxy S22
CPU	2 × ARM Cortex-X1 2 × ARM Cortex-A76 4 × ARM Cortex-A55	1 × ARM Cortex-X2 3 × ARM Cortex-A710 4 × ARM Cortex-A510
GPU	ARM Mali-G78	Qualcomm Adreno 730
Accelerator	Google Edge TPU	Qualcomm Hexagon
Release	October 2021	February 2022

B. HARDWARE CHARACTERIZATION

This study investigates the throughput improvement of deep learning applications on commercial mobile devices, specifically Google Pixel 6 and Samsung Galaxy S22. These devices feature not only CPU and GPU but also an AI accelerator. The detailed specifications for both mobile devices are provided in Table 1. While the accelerator efficiently processes operations, particularly convolutions, in the quantized network at high speeds, certain operations may be inefficient or may result in a slower overall network execution when falling back to CPU. It is crucial to thoroughly investigate the effective mapping of layers that the accelerator can process to achieve optimal throughput improvements.

Users are unable to customize settings, such as frequency and thread mapping, in commercial mobile devices due to restricted user permissions, which is a feature unlike embedded boards. For instance, Dynamic Voltage and Frequency Scaling (DVFS) governors on commercial mobile devices automatically regulate the processor's frequency based on processor utilization. Users typically do not have the authority to control the frequency on mobile devices. When running deep learning applications with multi-threading on the multi-core CPU, users cannot precisely specify which core to utilize; this is determined by the OS scheduler based on system conditions.

Thermal management plays a crucial role in mobile devices. Methods like frequency throttling, core idle

TABLE 2. Comparison with the related works on throughput enhancement. Acc. is an abbreviation for an accelerator.

Works	Platform	Acc. use	Quantization	Performance estimation	Inference body parallelization	Thermal analysis
D. Kang et al. [14]	Embedded board	X	X	Profiling	X	X
E. Jeong et al. [7]	Embedded board	O	PTQ, symmetric only	Measurement	Pipelineing, Streaming (GPU, Acc.)	X
DeepX [15]	Mobile device	X	X	Profiling	Network partitioning (CPU, GPU)	X
CNNdroid [16]	Mobile device	X	X	Profiling	Network partitioning (CPU, GPU)	X
μLayer [17]	Mobile device	X	QAT	Profiling	Single layer partitioning (CPU, GPU)	X
D. Kang et al. [18]	Mobile device	X	X	Profiling	Pipelining (CPU, GPU)	X
T. Tan et al. [19]	Mobile device	O	X	Profiling	Network partitioning (CPU, Acc.)	X
J. Kwon et al. [20]	Mobile device	O	X	Profiling	Pipelining (CPU and GPU combined, Acc.)	X
Proposed	Mobile device	O	PTQ, mixed specification	Measurement	Pipelining (CPU, GPU, Acc.)	O

injection, and core hotplug are employed to mitigate temperature and control heat. As a result, system performance gradually declines as the temperature increases. We need to address these constraints for performance optimization.

III. RELATED WORK

Pipelining is the basic method to improve the throughput performance of deep learning applications. On a simple heterogeneous system that consists of a CPU and a GPU, the throughput of a deep-learning application could be increased by simply separating the execution of pre-/post-processing on the CPU from the execution of compute-intensive inference body on the GPU [14]. Further speed-up is achieved by parallelizing pre-/post processing on the CPU and by using FP16 quantization and low-rank optimization techniques for the inference body on the GPU. But the inference body itself was not pipelined in this approach.

Many studies have been carried out to improve throughput in CPU-GPU systems. For instance, DeepX [15] partitions layers of a network based on layer-wise profiling of performance and energy consumption, compresses each partitioned segment, and then maps them to CPUs and GPUs to boost throughput. CNNdroid [16] improves processing efficiency by allocating data-parallel and compute-intensive layers such as convolution or fully connected layers to a GPU, while distributing the remaining layers across multiple CPUs. μLayer [17] utilizes quantization through QAT to optimize the network itself. For each layer, the performance of CPUs and GPUs is profiled. Based on this profiling information, channel-wise partitioning of each layer and distribution to CPUs and GPUs are performed, contributing to increased throughput.

While some research efforts have explored AI accelerators to improve throughput, these works predominantly focus on embedded boards rather than commercial mobile devices. They overlook the constraints imposed by the unique characteristics of processors within the mobile devices, as well as software factors such as OS schedulers and DVFS governors. Some works assume the exclusive use of GPUs and accelerators for the inference body, neglecting the potential for exploiting more fine-grained parallelism.

Kang et al. [18] attempted to pipeline a network in mobile devices equipped with accelerators, but limited software support for the accelerators hindered implementation on a real device. In their study, they enhanced throughput by utilizing CPUs and GPUs only by mapping layers onto the processors and pipelining them. Even when utilizing accelerators within mobile devices, Tan and Cao [19] accelerated deep learning using only CPUs and accelerators without considering GPUs. The use of CPUs is solely to map some layers to the CPUs, not to improve performance, but rather to reduce accuracy drops. J. Kwon et al. [20] proposed to pipeline deep learning networks utilizing CPUs, GPUs, and accelerators, configuring the pipeline with only two stages. One stage is managed by the accelerators, while the other stage is handled by the CPUs and/or GPUs. The lack of separation between the CPUs and GPUs as distinct stages constrains the granularity of pipelining, resulting in a performance gain of only around 2 times.

Table 2 provides a summary of the comparison between the proposed technique with related works. In contrast to most works that assume layer profiling on all processing elements can be done a priori, we do not make such an assumption that is often infeasible particularly for commercial mobile devices. Instead we estimate the performance by direct measurement after running the pipelined network on the mobile device.

The work most closely related to ours is the work targetting the NVIDIA Jetson board, considering both quantization and pipelining [7], but there are notable distinctions. Their approach lacked consideration for utilizing the CPU in the pipelined execution of the inference body, and they did not investigate the effects of thermal throttling in their study. They exclusively relied on symmetric quantization supported by TensorRT. Our work not only integrates quantization and pipelining but also emphasizes the role of CPU in the pipelined execution of the inference body. Our methodology extends beyond the NVIDIA Jetson board to commercial mobile devices, acknowledging the unique constraints imposed by mobile processors and the influence of software factors, including OS schedulers and DVFS governors. By conducting experiments on devices like Google Pixel 6 and Samsung Galaxy S22, we provide a practical and relevant evaluation that includes the impact of thermal throttling, contributing to a more comprehensive understanding of deep learning application optimization on commercial smartphones.

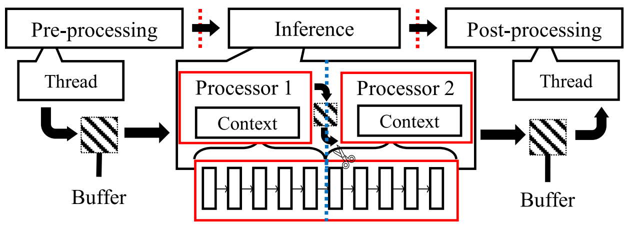


FIGURE 3. Overview of execution of pipelined deep learning application. Pipelining stages include not only pre-processing and post-processing but also partitioned multiple stages of the inference body. .

TABLE 3. Quantized operator specification to reduce accuracy drop.

Operator	Activation	Weight
Convolution	asymmetric per-tensor	symmetric per-channel
DW Convolution	asymmetric per-tensor	symmetric per-channel
Fully Connected	asymmetric per-tensor	symmetric per-tensor
Others	asymmetric per-tensor	-

IV. PROPOSED METHODOLOGY

The proposed approach for enhancing the throughput performance of a deep learning application comprises three key steps. The initial step involves quantizing network parameters pre-trained in FP32 to INT8, facilitating efficient utilization of deep learning accelerators. After the quantization process, we optimize the network by leveraging all available processing elements. In scenarios where the architectural structure of a deep learning network is non-linear, we serialize the layer execution, referring to the execution order among layers as determined by the associated SDK in the baseline implementation.

As illustrated in Figure 3, we not only pipeline the pre-/post-processing stages but also partition the inference stage into multiple stages, mapping them to heterogeneous processors for execution. The design space for pipelining is too wide to explore exhaustively. We employ a genetic algorithm as a meta-heuristic to find a solution close to optimal. The last step involves examining the effect of potential thermal throttling during continuous execution of the application.

A. POST TRAINING QUANTIZATION

We employ different quantization schemes for activation data and filter weights to minimize the accuracy drop caused by PTQ. As shown in Table 3, quantization settings are customized for each operator, incorporating combinations of symmetric and asymmetric quantization, as well as per-tensor and per-channel quantization. For activations, we choose asymmetric per-tensor quantization. This decision accounts for the varied distribution of FP32 values across the entire activation, and per-tensor quantization helps reduce quantization errors across the entire activation in comparison to channel-wise quantization. Considering the typically asymmetric distribution of activations, we permit the zero point to take any value from -128 to 127. This ensures that FP32 values are allocated with more representative bits, thereby improving accuracy [21].

Symmetric per-channel quantization is applied to the weights of filtering operators, including normal convolution and depthwise convolution. This decision is guided by the observed variation in weight distribution across channels for these operators, making per-channel quantization effective in minimizing quantization errors. We opt for symmetric quantization due to their more symmetric tendency compared to activations and to circumvent the computational inefficiency associated with considering different zero points for each channel during asymmetric quantization [21].

To execute quantization accurately, the data employed for calibration in Post-Training Quantization (PTQ) undergoes normalization and resizing. The calibration process involves a selection of 1000 data points randomly chosen from the training dataset, ensuring a robust representation for effective calibration.

B. PRE-/POST-PROCESSING PIPELINING

To enable parallel execution of pre-processing and post-processing stages in deep learning applications, we designate them as separate pipelining stages. In the pre-processing stage, operations such as loading images and resizing are performed to prepare the input for inference. In the post-processing stage, the output tensor obtained from the inference is processed to generate the final output, revealing the application-specific results. For pre-processing and post-processing, 2 threads are allocated, as it is observed that overall performance reaches a plateau beyond 2 threads. Allocating the minimum number of threads for pre-processing helps mitigate competition with inference threads on the CPU. The number of pre-processing and post-processing threads is set to minimal. Pipelining the pre-processing and post-processing stages enables them to be executed concurrently with inference, significantly contributing to increased throughput.

C. INFERENCE PIPELINING AND OPTIMAL MAPPING SEARCH

The inference body is also divided into multiple pipeline stages, each mapped to distinct processing elements within the system. The primary challenge in inference pipelining lies in determining where to partition the inference body and how many stages to make. Achieving an optimal balance in the execution time of all pipeline stages is crucial for maximizing throughput performance. Suppose that the inference body consists of N layers and the system has 3 processing elements. The number of potential pipelining candidates becomes ${}_3C_N \times 3!$, which is too large to explore exhaustively. The pipelining technique in our study involves only one-to-one communication between processor stages, while optimization in heterogeneous systems considers various communication methods among processors.

The mapping exploration is conducted using a genetic algorithm, similar to the method described in [7].

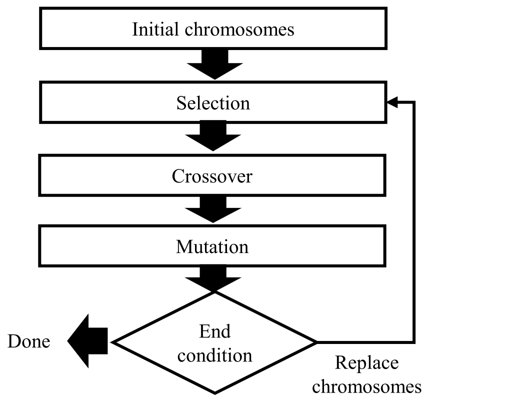


FIGURE 4. Genetic algorithm loop to explore the optimal mapping.

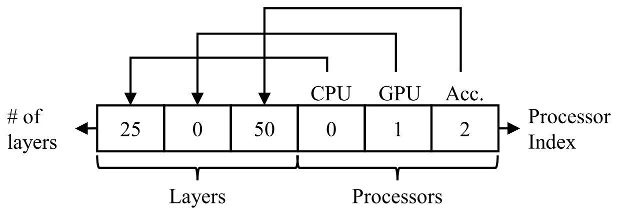


FIGURE 5. Structure of the chromosome to represent mappings for inference pipelining.

On commercial mobile devices, the operating frequency is determined by the DVFS governor beyond user control. The OS scheduler dictates the processor mapping, preventing users from controlling the mapping to a specific core, resulting in noticeable execution time variability. Establishing an accurate performance prediction model is challenging under these constraints. The proposed genetic algorithm explores the optimal mapping based on the measurement of actual system behavior when processing deep learning applications on mobile devices. This involves representing each pipelining candidate as a chromosome and utilizing genetic operations such as crossover and mutation to find chromosomes with proximity to the optimal fitness value. Figure 4 illustrates this search process.

Figure 5 describes the chromosome structure designed for mapping optimization, accommodating a maximum of three pipeline stages for the inference body. The first three genes represent the number of layers assigned to each pipeline stage while the next three genes specify the type of processor to which each pipeline stage is mapped. In the example provided in Figure 5, a chromosome is generated by the genetic algorithm for a deep learning network comprising 75 layers. The first 25 network layers are mapped to the CPU in the first stage. When the number of layers allocated to a pipeline stage is 0, the processor is not used for mapping. This chromosome implies a CPU-GPU pipelining without using an accelerator. The GPU is assigned layers 26 to 75.

When mapped to the CPU, it operates with 4 threads, taking into account the number of pre-processing and post-processing threads as well as CPU configuration of mobile devices. Two mobile devices used in this work have 8 CPU cores. If the number of threads for inference exceeds 4, more than two threads might run concurrently on the same core, introducing competition and potentially leading to performance degradation.

In the crossover process of these chromosomes, the genes corresponding to the number of layer allocations (the first three genes) and processor mapping (the last three genes) for two chromosomes are exchanged. The number of layer allocations or processor mappings in the mutation process is randomly altered. Crossover occurred with a 90% probability during chromosome generation, and mutation occurred with an 80% probability. As the probabilities of crossover and mutation increase, the likelihood of generating new chromosomes at each iteration in the genetic algorithm rises, while the convergence speed toward the optimal chromosome may slow down due to the necessity of considering a larger pool of candidates. We intentionally sacrifice some convergence speed to thoroughly explore a diverse range of chromosomes. This deliberate trade-off is chosen to maximize the throughput of deep learning applications by conducting a comprehensive examination of numerous chromosomes. The decision to use high probabilities for crossover and mutation aims to enhance the algorithm's ability to discover optimal mapping solutions.

The evaluation metric of the genetic algorithm is the throughput of the deep learning application, measured on an actual mobile device. The genetic algorithm is executed on the host PC to find a suboptimal pipelining solution, which is then transmitted to the mobile device. The mobile device executes the actual deep learning application with the requested mapping, and the measured throughput is sent back to the host PC. The exploration is conducted for a specified number of iterations as a hyper-parameter. Termination occurs if the top-performing 20 chromosomes remain unchanged for 10 consecutive iterations, allowing for sufficient exploration while enabling early stopping in case of convergence. All these processes are automated, facilitating the exploration of optimal mapping using actual measured data that accurately reflects the constraints within a mobile device. The exploration is conducted before the deployment of mapping solutions. After deployment, inference is performed using the discovered solution, ensuring no impact on runtime performance.

D. COMMUNICATION BETWEEN PIPELINED STAGES

In the optimization of mapping on heterogeneous systems, it is necessary to account for communication overhead caused by various communication patterns between processors, such as one-to-many, many-to-one, and so forth [22]. In the proposed pipelining technique, each stage communicates with the next stage in a one-to-one communication between processor stages. Our methodology proposes a measurement-based mapping exploration solution. In the measurement process, not only the processing time of each processor but also the communication between processors

TABLE 4. Benchmark object detection networks used in experiments.

Network	Label	Input size	# of layers
SSD MobileNet V1 FPN [27]	MV1	640×640	110
SSD MobileNet V2 FPN [27]	MV2	640×640	149
SSD ResNet50 FPN [28]	R50	640×640	154

is measured. This implies that communication overhead is considered during the exploration process, providing a more holistic assessment.

V. EXPERIMENTS

A. SETUP

To assess the performance of the proposed method, we measured the accuracy (average precision, AP) and throughput (frames per second, FPS) of an object detection application using 5,000 images in the validation set of the COCO2017 dataset. Quantization was conducted using MMDetection and MMRazor frameworks [23], [24] based on PyTorch and the PTQ API provided by TensorFlow Lite [25]. The calibration dataset comprises 1,000 randomly selected images from the COCO2017 train set. Network pipelining was implemented in TensorFlow Lite using the Coral library [2]. When processing layers using the GPU or the accelerator, unsupported operations are replaced with CPU execution by the deep learning library as a fallback. The genetic algorithm was implemented using the DEAP library [26].

Three object detection networks serve as benchmark applications, as outlined in Table 4. The input image size for each network is uniformly set to 640 × 640. In the case of SSD MobileNetV2 FPN, MobileNet V2 is used as the backbone with depthwise convolution operators and Feature Pyramid Network is used as the detection head. The maximum number of inference stages is limited to 3, aligning with the three heterogeneous processors in the target system, to avoid allocating more than one inference stage on the same processing element. The number of chromosomes used in the genetic algorithm is set to 32.

B. QUANTIZATION RESULTS

Different quantization schemes are applied to activation and weight values. This decision is based on preliminary experiments that compared the various combinations of quantization schemes, using MMDetection and MMRazor frameworks. While TensorFlow Lite allows for changing per-tensor and per-channel configurations through its API, there is no API support for modifying symmetric and asymmetric configurations. We manually explore various possible combinations as shown in Table 5. In this experiment, SSD ResNet50 FPN was used. For activation, per-channel quantization was not considered due to the lack of support in the framework.

Table 5 reveals that asymmetric quantization of activation leads to higher accuracy compared to symmetric quantization. This outcome can be attributed to the asymmetric distribution of activation, allowing the zero-point to have INT8 values other than 0, providing a better representation of FP32 values in INT8. With activation quantized as asymmetrical per-tensor, weights exhibit comparable accuracy levels when quantized as symmetrical or asymmetrical. However, there is a noticeable difference between per-tensor quantization and per-channel quantization. Hence, symmetric per-channel quantization is selected for weights to minimize the accuracy loss caused by quantization.

Table 6 presents the accuracy of each network after quantization. It is noteworthy that the accuracy of SSD MobileNet V1 FPN and SSD MobileNet V2 FPN is increased even after quantization. Quantization leads to a loss in accuracy due to precision reduction. In some cases, this loss may be compensated by the normalization effect caused by precision reduction, resulting in improved accuracy. In the case of SSD ResNet50 FPN, there is a negligible decrease in accuracy.

The weights of the networks used in TensorFlow Lite are stored in the tflite format. Figure 6 shows the changes in

TABLE 5. Accuracy variation in SSD ResNet50 FPN by varying quantization schemes.

Activation	Weight	AP_{50}	AP_{75}	$AP_{50:95}$
FP32	FP32	56.7%	39.6%	37.4%
symmetric per-tensor	symmetric per-tensor	55.8%	38.7%	36.6%
	symmetric per-channel	55.9%	39.2%	36.8%
	asymmetric per-tensor	55.2%	38.3%	36.1%
	asymmetric per-channel	56.1%	39.2%	36.8%
asymmetric per-tensor	symmetric per-tensor	56.2%	39.4%	37.1%
	symmetric per-channel	**56.3%**	**39.7%**	**37.2%**
	asymmetric per-tensor	55.6%	38.7%	36.5%
	asymmetric per-channel	**56.3%**	39.6%	**37.2%**
per-channel	-	Not Supported		

TABLE 6. Accuracy comparison between FP32 and INT8 networks using PTQ API provided by TensorFlow Lite.

Label	Precision	AP_{50}	AP_{75}	$AP_{50:95}$
MV1	FP32	44.4%	33.9%	30.9%
	INT8	47.5%	34.7%	32.2%
MV2	FP32	45.6%	32.2%	30.6%
	INT8	45.9%	33.6%	31.1%
R50	FP32	54.6%	44.6%	40.1%
	INT8	54.7%	43.8%	39.6%

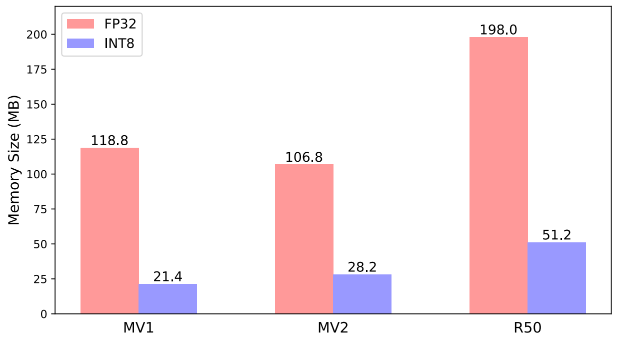


FIGURE 6. Comparing memory sizes of FP32 and INT8 networks in TFLite format.

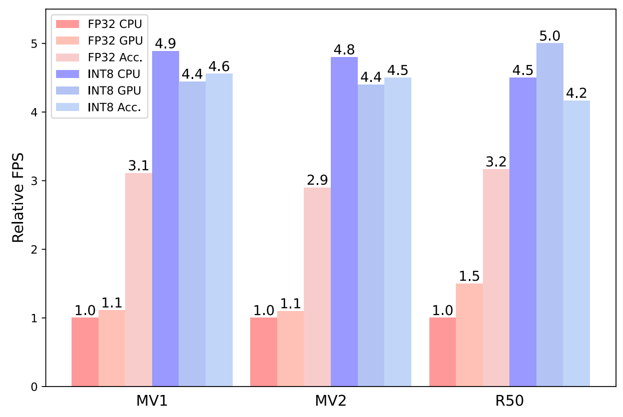


FIGURE 7. Comparing relative FPS of FP32 and INT8 networks through single processor inference mappings on Pixel 6.

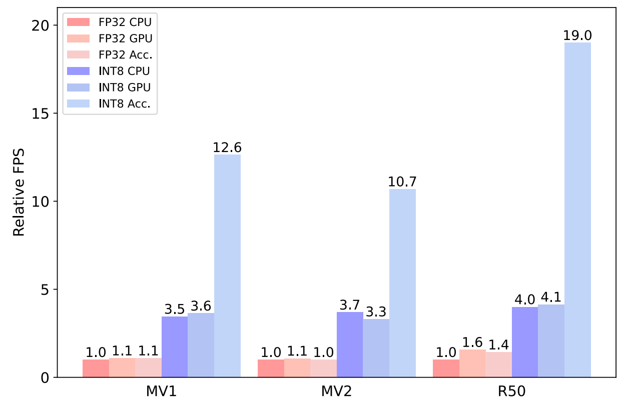


FIGURE 8. Comparing relative FPS of FP32 and INT8 networks through single processor inference mappings on Galaxy S22.

memory size before and after quantization. Each network is converted to a tflite file that is 18%, 26.4%, and 25.9% of the respective FP32 networks' memory size. While a reduction rate of about 25% is expected, SSD MobileNet V1 FPN experiences a more substantial reduction in memory size compared to other networks. This can be attributed to Tensorflow Lite that optimizes the network extensively during quantization.

Figures 7 and 8 depicts the comparison of relative FPS between FP32 and INT8 networks after quantization. The relative values are computed using the FPS of FP32 inference mapping on the CPU as the reference. The network execution comprises separate threads for pre-processing, inference, and post-processing. Pre-processing and post-processing are handled by the CPU, while the whole inference body is executed on each processing element: CPU, GPU, or AI accelerator.

On Pixel 6, comparing the throughput for each processor type reveals a $\times 4.8$ to $\times 5$ improvement for CPU and a $\times 3.3$ to $\times 4$ improvement for GPU. The improvement in FPS due to quantization is less significant in the AI accelerator compared to CPUs and GPUs, as the AI accelerator's FPS remains high even for FP32 networks. The AI accelerator in Pixel 6 is an

TABLE 7. Pipelining solutions obtained by the proposed technique based on genetic algorithm.

Device	Label	Iteration	Mapping order	# of Layers
Pixel 6	MV1	200	GPU - Acc. - CPU	{40, 44, 26}
	MV2	300	GPU - Acc. - CPU	{79, 38, 32}
	R50	300	Acc. - GPU - CPU	{75, 24, 55}
Galaxy S22	MV1	200	Acc. - GPU - CPU	{102, 5, 3}
	MV2	300	Acc. - GPU - CPU	{143, 2, 4}
	R50	300	Acc. - GPU - CPU	{142, 7, 5}

TABLE 8. The search time and the proportion of explored design space to obtain optimal pipelining solutions.

Device	Label	Search time (hours)	Searched solutions	Explored design space (%)
Pixel 6	MV1	11.3	1571	4.45
	MV2	19.3	3016	4.62
	R50	34.2	3061	4.39
Galaxy S22	MV1	3.6	801	2.25
	MV2	10.6	2554	3.89
	R50	19.4	3643	5.19

edge TPU, and software optimizations for FP32 networks seem to be effective, providing better FP32 performance than the other processors. The performance improvement from quantization in the AI Accelerator on Pixel 6 ranges from $\times 1.3$ to $\times 1.5$.

On Galaxy S22, quantization results in performance improvements ranging from $\times 3.5$ to $\times 4$ for CPU and $\times 2.5$ to $\times 3.2$ for GPU. Unlike the case of Pixel 6, a notable increase ranging from $\times$ 10.9 to $\times$ 13.5 is observed in the AI accelerator. This impressive performance is attributed to the higher INT8 computation speed of the Qualcomm Hexagon AI accelerator in Galaxy S22 compared to CPU and GPU. The performance gap among processors has implications for inference pipelining and optimal mapping searches, leading to different execution mappings between the two smartphones.

C. PIPELINING AND MAPPING EXPLORATION

Table 7 presents the results of the optimal mapping exploration for each network obtained through the genetic algorithm on Pixel 6 and Galaxy S22. The maximum number of iterations is varied according to the number of layers in each network, allowing for more extensive exploration with a higher number of layers. The mapping of network layers is performed by dividing them based on the order of processors' mapping sequence, starting from the first layer and allocating layers in ascending order.

Table 8 shows the search time and the explored design space for finding optimal pipelining solutions. The search time is derived from measurements, influenced primarily by the performance of the mobile devices executing the application rather than the host computer that runs the GA algorithm. The searched solutions pertain to the distinct solutions examined with previously explored chromosomes being reused during exploration when regenerated. The explored design space indicates the ratio of examined solutions to

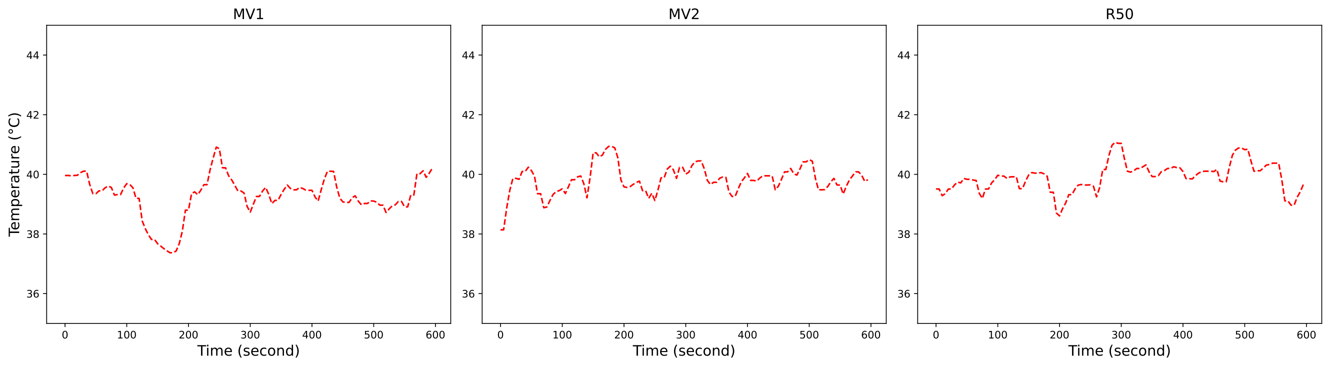


FIGURE 9. Device surface temperature variation trends when exploring optimal mapping solution on Pixel 6.

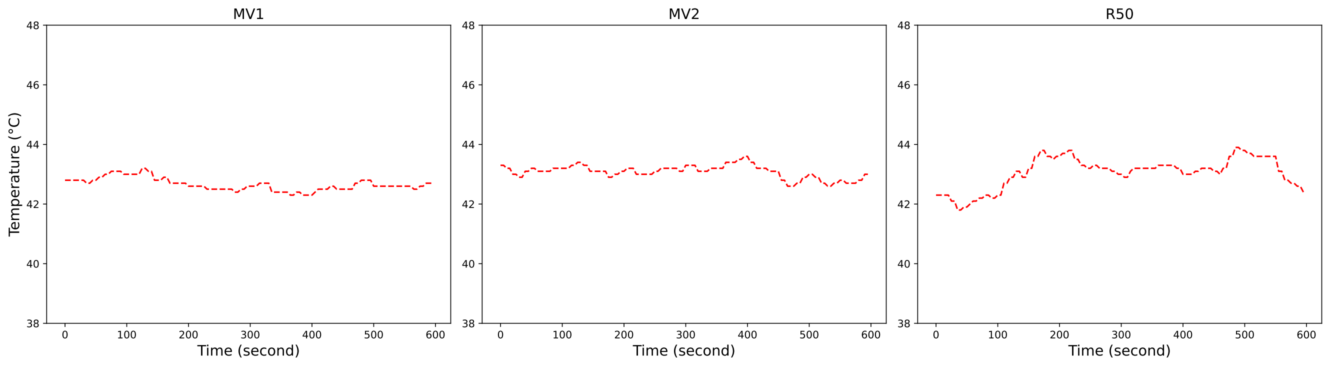


FIGURE 10. Device surface temperature variation trends when exploring optimal mapping solution on Galaxy S22.

all possible solutions for each network. The search time varies depending on the performance of the mobile device and network characteristics. Upon examining the explored design space, it is observed that, with the exception of SDD ResNet50 FPN on Galaxy S22, all solutions explored cover less than 5% of the design space. This demonstrates that the proposed method can find optimal pipelining solutions efficiently.

Figures 9 and 10 illustrate the device surface temperatures measured during the execution of the genetic algorithm for optimal mapping exploration. The Android Hardware Abstraction Layer (HAL) thermal service facilitated continuous monitoring, starting one hour after the genetic algorithm initiation and lasting for 10 minutes. Despite fluctuations tied to the explored solutions, Pixel 6 maintained a temperature of around 40 degrees, and Galaxy S22 hovered around 43 degrees. These temperatures align with the thermal throttling range of commercial mobile devices. These results highlight the occurrence of thermal throttling during the mapping exploration, which in turn influenced the measured throughput and decisions made by the genetic algorithm. In short, thermal throttling is considered during mapping in the proposed methodology.

Figures 11 and 12 compare the FPS results of mapping solutions obtained through optimal mapping search with the maximum values when mapping FP32 and INT8 networks to a single type of processor on Pixel 6 and Galaxy S22,

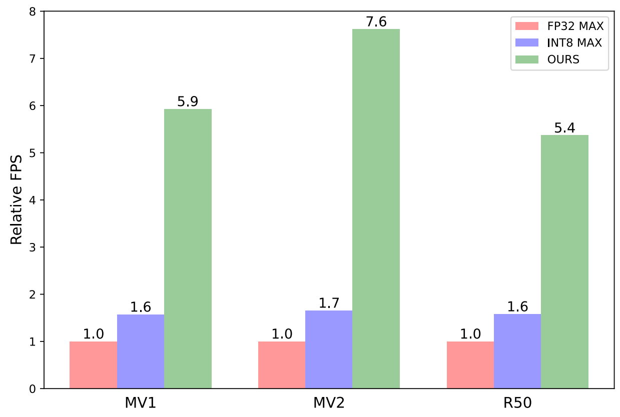


FIGURE 11. Comparing relative FPS between single processor mappings and optimal pipelined mapping on Pixel 6.

respectively. Relative FPS is calculated with reference to the maximum FPS from single processor mappings of FP32 inference, processing 5,000 images; the maximum FPS for single processor mappings of the FP32 network is achieved when it is mapped to the accelerator in case of Pixel 6 and to the GPU in case of Galaxy 22. On Pixel 6, FPS from optimal pipelining mapping is ×3.3 to ×4.4 higher than the maximum FPS of INT8 inference mapping to a single processor and ×5.4 to ×7.6 higher than the maximum FPS of FP32 inference. On Galaxy S22, compared to INT8 single

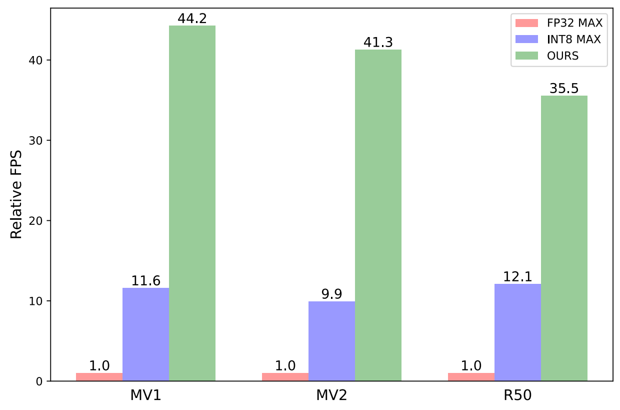


FIGURE 12. Comparing relative FPS between single processor mappings and optimal pipelined mapping on Galaxy S22.

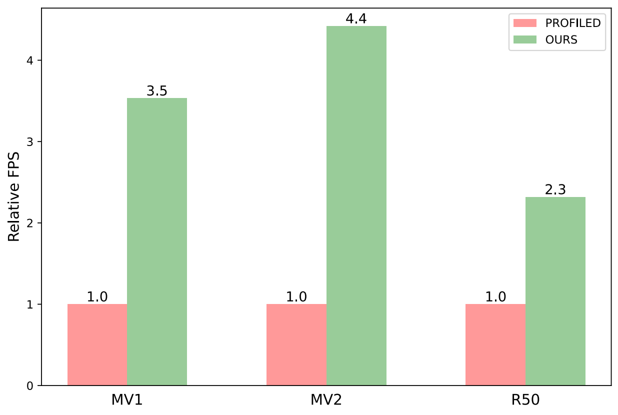


FIGURE 13. Comparing relative FPS of the profiling-based and the proposed methods on Pixel 6.

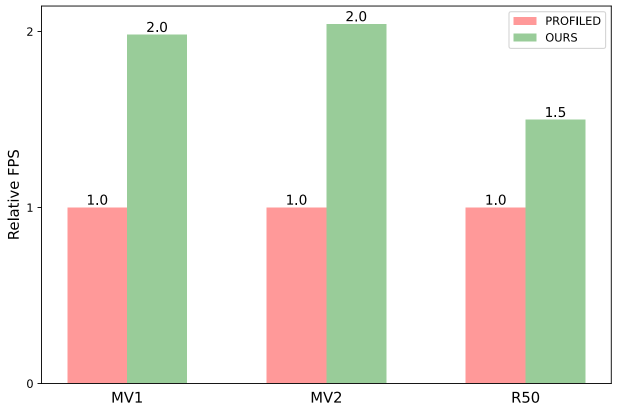


FIGURE 14. Comparing relative FPS of the profiling-based and the proposed methods on Galaxy S22.

processor mapping, FPS has improved from ×2.9 to ×4.1, and compared to FP32 single processor mappings, FPS has improved from ×35.5 to ×44.2 This indicates that dividing the inference stage among multiple processors during network execution can significantly increase throughput.

It is noteworthy that the sub-optimal mapping order found by the genetic algorithm differs between the two smartphones. For Pixel 6, the mapping order depends also on the benchmark networks; organizing the pipeline in the sequence of GPU-accelerator-CPU is optimal for SSD MobileNet V1 FPN and SSD MobileNet V2 FPN while the optimal order is found to be accelerator-GPU-CPU for SSD ResNet 50 FPN. This variation in the exploration of processor execution order for each network may be caused by the operating mechanism of the genetic algorithm, evolving chromosomes based on the processing order that shows good throughput in the early iterations. CPU consistently appears at the end of the execution order. This is attributed to CPU fallback of GPU and accelerator occurring in the later layers of the networks. Furthermore, a substantial number of layers are mapped to CPU since the CPU performance is comparable to GPU or AI accelerator in smartphones, as shown in Figure 7.

On Galaxy S22, the mapping order is the same for all networks. Another notable difference from Pixel 6 lies in the number of mapped layers for each processing element, with most layers mapped to the accelerator. This aligns with the earlier observations of the accelerator performance for the INT8 network after quantization in Figure 12.

These experiments confirm that the proposed approach demonstrates adaptability in customizing the mapping strategy based on the characteristics of the network and processors in the system. This underscores the effectiveness of the proposed method in optimizing network execution across diverse processors.

D. COMPARISON WITH PROFILING-BASED MAPPING SOLUTIONS

An experiment has been conducted to compare the proposed technique with profiling-based methods that are widely adopted in many related works. Conducting individual layer profiling on commercial mobile devices is impractical due to the lack of support from Tensorflow Lite and chip vendors. To overcome these constraints, we estimated the profiled execution time for each layer by assuming that the execution time is proportional to the computational demand, or the number of required FLOPs.

We defined the processor's performance based on the single-processor inference mapping performance for each network and derived mapping solutions accordingly. The single-processor inference mapping performance has already been demonstrated in Figures 7 and 8, and we used INT8 inference results as the processor performances. We defined the computational demand for each layer based on the required FLOPs of convolution layers within the network, adjusting the workload according to the performance ratio of each processor in Table 9. The exploration of mapping solutions was conducted for all permutations of the three processors, considering scenarios where all three types of processors are used. We partitioned the network into three parts and adjusted the number of layers in each partition until a balance in the computational workload across the three parts was achieved. Among the obtained mapping

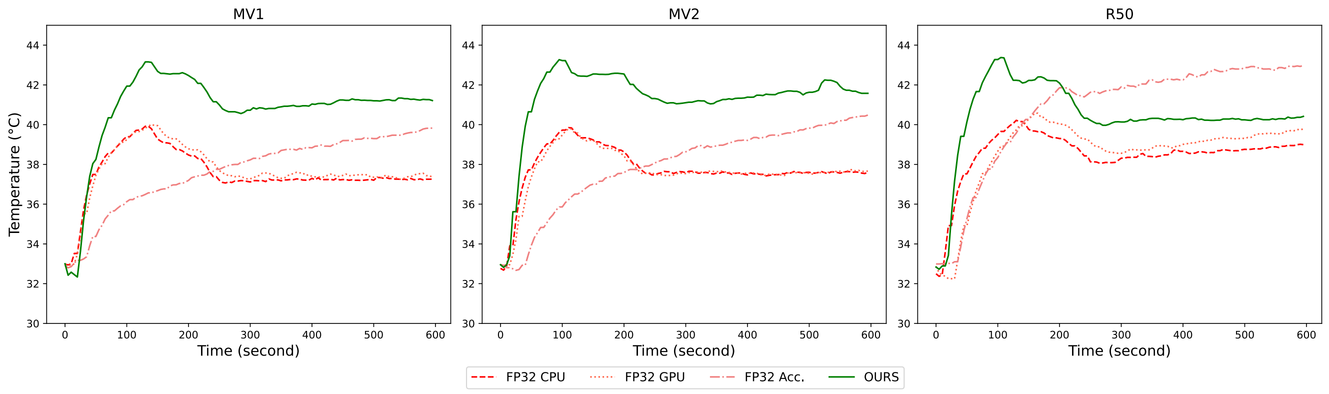


FIGURE 15. Device surface temperature variation trends when running deep learning networks on Pixel 6.

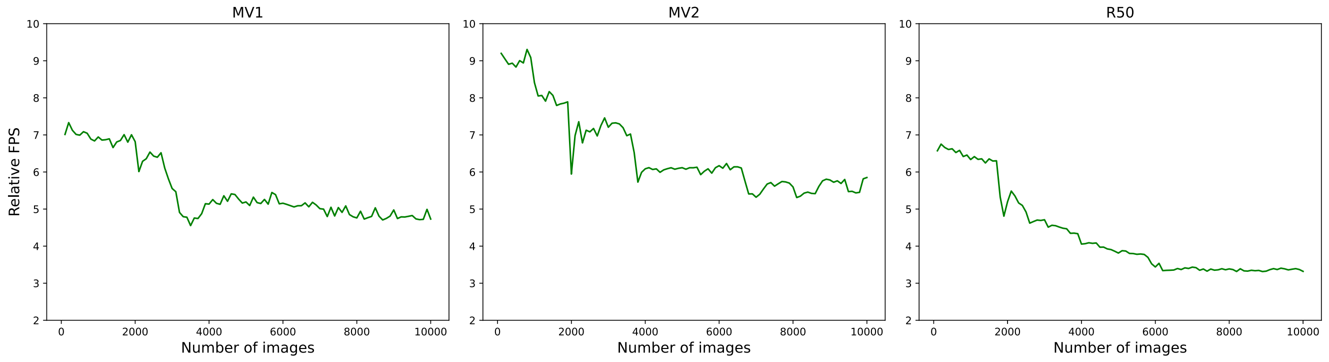


FIGURE 16. Throughput variations over 100 image intervals for processing 10,000 images on Pixel 6.

TABLE 9. The performance ratio of each processor based on INT8 single processor mapping inference throughput.

Device	Label	CPU ratio	GPU ratio	Acc. ratio
Pixel 6	MV1	1.0	0.897	0.938
	MV2	1.0	0.916	0.937
	R50	1.0	1.111	0.933
Galaxy S22	MV1	1.0	1.028	3.6
	MV2	1.0	0.891	2.891
	R50	1.0	1.025	4.75

TABLE 10. Pipelining solutions obtained by the profiling-based method.

Device	Label	Mapping order	# of Layers
Pixel 6	MV1	GPU - CPU - Acc.	{50, 16, 44}
	MV2	Acc. - GPU - CPU	{95, 20, 34}
	R50	Acc. - GPU - CPU	{56, 48, 50}
Galaxy S22	MV1	Acc. - GPU - CPU	{66, 10, 35}
	MV2	Acc. - GPU - CPU	{111, 11, 28}
	R50	Acc. - GPU - CPU	{105, 10, 40}

solutions, we selected the one with the highest throughput for comparison. The optimal solutions obtained by the profiling-based method are displayed in Table 10

Figures 13 and 14 compare the throughput evaluation results between the profiling-based method and the mapping solutions found by our methodology. In Pixel 6, our approach exhibited ×2.3 to ×4.4 higher throughput, while in Galaxy S22, it showed ×1.5 to ×2 higher throughput. These results signify that relying solely on processor performance-based network partitioning methodologies does not guarantee optimal pipelining performance. Our methodology, guided by the empirical measurement-based genetic algorithm, explores the design space, allowing us to consider various factors such as communication, CPU fallback, and more, beyond just processor performance, to find the optimal mapping.

E. THERMAL THROTTLING IMPACT

The proposed method utilizes all processor elements within the mobile devices and maximizes their utilization to enhance pipelining performance. It will increase the device's temperature more rapidly compared to single processor mappings. The thermal throttling mechanism adopted in the device may have a significant impact on system performance. It is essential to investigate the impact of thermal throttling on throughput performance of the mobile devices. Thermal throttling leads to a gradual decline in system performance, eventually reaching a point of convergence.

Figures 15, 16, 17, and 18 depict the trends in device surface temperature and throughput variations as the number of processed images increases when applying our methodology to each mobile device. To measure temperature, each device was left at room temperature until it reached 33 degrees Celsius, and the temperature change was recorded every

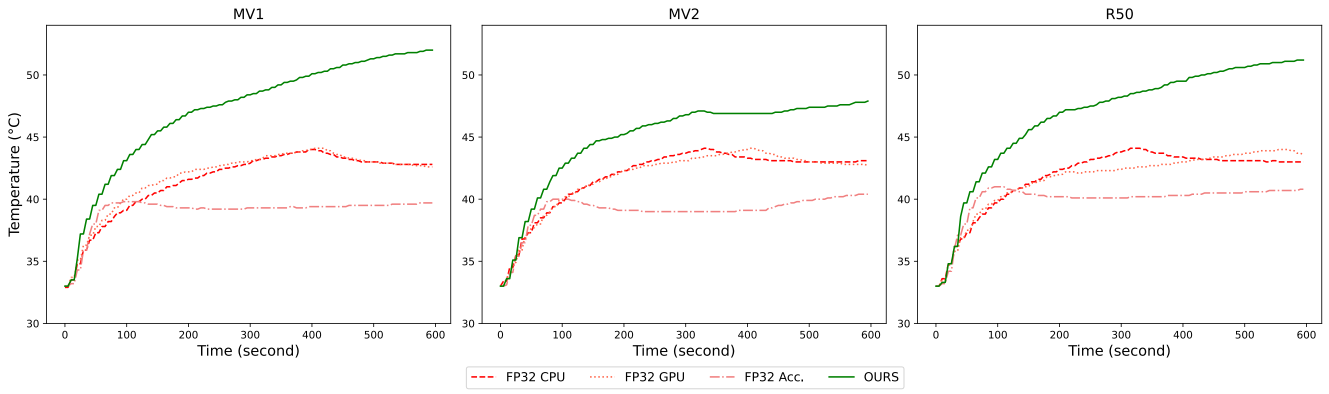


FIGURE 17. Device surface temperature variation trends when running deep learning networks on Galaxy S22.

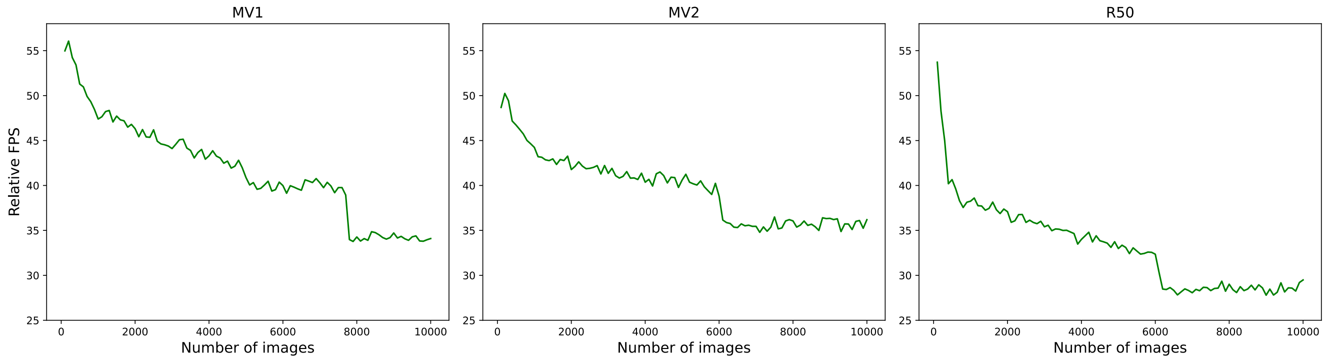


FIGURE 18. Throughput variations over 100 image intervals for processing 10,000 images on Galaxy S22.

5 seconds when the deep learning network continued to run for 10 minutes after reaching 33 degrees Celsius. Throughput variation was assessed by inputting 10,000 images into the pipeline explored by the proposed method, calculating the throughput for every 100 images, and measuring the FPS relative to the maximum throughput of the FP32 network.

As shown in Figure 15, the temperature of Pixel 6 steadily increases initially, subsequently begins to decrease, and stabilizes below a certain level after about 300 seconds. Figure 16 shows the throughput variation that initially demonstrates high throughput, gradually decreases, and eventually stabilizes. The stabilized performance persists at a level surpassing both the maximum throughput of the FP32 network and the maximum throughput of the INT8 network.

In the case of Galaxy S22, as depicted in Figures 17 and 18, the temperature rises significantly higher than in Pixel 6. When using CPU or GPU alone for inference processing, thermal throttling could be observed, causing the device to operate under 40 degrees Celsius. When the accelerator is used for pipelined processing, the temperature increases to much a higher level although the rate of temperature increase gradually diminishes. Examining the FPS variation graph in Figure 18, it appears that thermal throttling is also applied to the accelerator to stabilize the FPS performance eventually. The stabilized performance is still significantly higher than the maximum FPS performance of the FP32 network on GPU.

Thermal throttling decreases the throughput performance over time during the execution of the deep learning application, eventually stabilizing the performance. The thermal throttling mechanism exhibits a wide variation between devices and even among processing elements in the same device. Despite thermal throttling induced by temperature rise, the proposed technique significantly boosts the FPS performance over the baseline single processor execution. If we can manage the DVFS policy for each processing element at the user level, it could be an interesting future research topic to consider the voltage and frequency settings as well as the mapping of layers for sustainable throughput maximization.

VI. CONCLUSION

In this paper, a novel methodology is introduced to optimize the throughput of deep learning applications on commercial smartphones that integrate dedicated AI accelerators alongside CPUs and GPUs. The proposed technique involves quantizing network parameters from FP32 to INT8, partitioning and mapping the deep learning networks across different processors and employing pipelining to enhance throughput.

Exploring the mapping design space, facilitated by a genetic algorithm, is a distinctive feature of our study, allowing us to identify the best mapping solutions tailored to the unique characteristics of each commercial smartphone.

This adaptive approach, combined with the precise measurement of actual execution times, considering dynamic voltage and frequency scaling (DVFS) and OS scheduling, represents a significant departure from conventional profiling methods. The impact of thermal throttling on throughput is investigated while exploring optimal mapping solutions and running benchmark applications continuously for 10 minutes. With this proposed technique, we achieve a remarkable $\times 5.4$ to $\times 7.6$ throughput improvement for various networks on Google Pixel 6 and $\times 35.5$ to $\times 44.2$ on Samsung Galaxy S22, compared to mapping FP32 networks on a single type of processor.

The significance of our work extends beyond experimental results; it challenges the conventional reliance on layer-wise profiling and offers a concrete, measurement-based alternative. This study not only contributes to the scientific understanding of deep learning application optimization on commercial smartphones but also enables more efficient and applicable methodologies in this domain.

REFERENCES

[1] *NVIDIA Jetson AGX Xavier Developer Kit*. Accessed: Nov. 10, 2023. [Online]. Available: https://developer.nvidia.com/embedded/jetson-agx-xavier-developer-kit

[2] *Google Coral*. Accessed: Dec. 12, 2023. [Online]. Available: https://coral.ai

[3] *Google Pixel 6 Smartphone*. Accessed: Dec. 12, 2023. [Online]. Available: https://support.google.com/pixelphone/answer/7158570?hl=e

[4] *Samsung Galaxy S22 Smartphone*. Accessed: Dec. 12, 2023. [Online]. Available: https://www.samsung.com/my/smartphones/galaxy-s22/specs

[5] E. Park, J. Ahn, and S. Yoo, "Weighted-entropy-based quantization for deep neural networks," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit. (CVPR)*, Jul. 2017, pp. 7197–7205.

[6] H. Li, A. Kadav, I. Durdanovic, H. Samet, and H. Peter Graf, "Pruning filters for efficient ConvNets," 2016, *arXiv:1608.08710*.

[7] E. Jeong, J. Kim, S. Tan, J. Lee, and S. Ha, "Deep learning inference parallelization on heterogeneous processors with TensorRT," *IEEE Embedded Syst. Lett.*, vol. 14, no. 1, pp. 15–18, Mar. 2022.

[8] A. Gholami, S. Kim, Z. Dong, Z. Yao, M. W. Mahoney, and K. Keutzer, "Book chapter: A survey of quantization methods for efficient neural network inference," in *Low Power Computer Vision*. Boca Raton, FL, USA: CRC Press, Feb. 2022, pp. 291–402.

[9] 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.

[10] W. Sung, S. Shin, and K. Hwang, "Resiliency of deep neural networks under quantization," 2015, *arXiv:1511.06488*.

[11] M. Nagel, R. A. Amjad, M. van Baalen, C. Louizos, and T. Blankevoort, "Up or down? Adaptive rounding for post-training quantization," in *Proc. Int. Conf. Machine. Learn. (ICML)*, 2020, pp. 7197–7206.

[12] M. Nagel, M. V. Baalen, T. Blankevoort, and M. Welling, "Data-free quantization through weight equalization and bias correction," in *Proc. IEEE/CVF Int. Conf. Comput. Vis. (ICCV)*, Oct. 2019, pp. 1325–1334.

[13] M. Nagel, M. Fournarakis, R. Ali Amjad, Y. Bondarenko, M. van Baalen, and T. Blankevoort, "A white paper on neural network quantization," 2021, *arXiv:2106.08295*.

[14] D. Kang, D. Kang, J. Kang, S. Yoo, and S. Ha, "Joint optimization of speed, accuracy, and energy for embedded image recognition systems," in *Proc. Design, Autom. Test Eur. Conf. Exhibition (DATE)*, Mar. 2018, pp. 715–720.

[15] N. D. Lane, S. Bhattacharya, P. Georgiev, C. Forlivesi, L. Jiao, L. Qendro, and F. Kawsar, "DeepX: A software accelerator for low-power deep learning inference on mobile devices," in *Proc. 15th ACM/IEEE Int. Conf. Inf. Process. Sensor Netw. (IPSN)*, Apr. 2016, pp. 1–12.

[16] S. S. Latifi Oskouei, H. Golestani, M. Hashemi, and S. Ghiasi, "CNNdroid: GPU-accelerated execution of trained deep convolutional neural networks on Android," in *Proc. 24th ACM Int. Conf. Multimedia*, Oct. 2016, pp. 1201–1205.

[17] Y. Kim, J. Kim, D. Chae, D. Kim, and J. Kim, "μLayer: Low latency on-device inference using cooperative single-layer acceleration and processor-friendly quantization," in *Proc. 14th EuroSys Conf.*, Mar. 2019, p. 115.

[18] D. Kang, J. Oh, J. Choi, Y. Yi, and S. Ha, "Scheduling of deep learning applications onto heterogeneous processors in an embedded device," *IEEE Access*, vol. 8, pp. 43980–43991, 2020.

[19] T. Tan and G. Cao, "Deep learning on mobile devices with neural processing units," *Computer*, vol. 56, no. 8, pp. 48–57, Aug. 2023.

[20] J. Kwon, J. Lee, and H. Kim, "Pipelining of a mobile SoC and an external NPU for accelerating CNN inference," *IEEE Embedded Syst. Lett.*, early access, Aug. 14, 2024.

[21] H. Wu, P. Judd, X. Zhang, M. Isaev, and P. Micikevicius, "Integer quantization for deep learning inference: Principles and empirical evaluation," 2020, *arXiv:2004.09602*.

[22] M. F. Reza, D. Zhao, and M. Bayoumi, "Energy-efficient task-resource co-allocation and heterogeneous multi-core NoC design in dark silicon era," *Microprocessors Microsyst.*, vol. 86, Oct. 2021, Art. no. 104055.

[23] K. Chen et al., "MMDetection: Open MMLab detection toolbox and benchmark," 2019, *arXiv:1906.07155*.

[24] *OpenMMLab Model Compress. Toolbox Benchmark*. Accessed: Dec. 17, 2023. [Online]. Available: https://github.com/open-mmlab/mmrazor

[25] M. Abadi et al., "TensorFlow: Large-scale machine learning on heterogeneous distributed systems," 2016, *arXiv:1603.04467*.

[26] F.-M. De Rainville, F.-A. Fortin, M.-A. Gardner, M. Parizeau, and C. Gagné, "DEAP: A Python framework for evolutionary algorithms," in *Proc. 14th Annu. Conf. Companion Genetic Evol. Comput.*, Jul. 2012, pp. 85–92.

[27] 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*.

[28] T.-Y. Lin, P. Dollár, R. Girshick, K. He, B. Hariharan, and S. Belongie, "Feature pyramid networks for object detection," 2016, *arXiv:1612.03144*.

CHOONGHOON PARK received the B.S. degree in computer science and engineering from Seoul National University, Seoul, South Korea, in 2017, where he is currently pursuing the M.S. degree in computer science and engineering. His current research interests include embedded machine learning and network optimization on embedded systems.

SOONHOI HA (Fellow, IEEE) received the B.S. and M.S. degrees in electronics engineering from Seoul National University, Seoul, South Korea, in 1985 and 1987, respectively, and the Ph.D. degree in electrical engineering and computer science from the University of California at Berkeley, Berkeley, CA, USA, in 1992. He is currently a Professor with Seoul National University. His current research interests include the HW/SW co-design of embedded systems, embedded machine learning, and the Internet of Things. He has actively participated in the premier international conferences in the EDA area, for instance serving ESWEEK, in 2018, as the General Chair. He is a member of ACM.