IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Received October 30, 2021, accepted December 1, 2021, date of publication December 6, 2021,
date of current version December 20, 2021.

Digital Object Identifier 10.1109/ACCESS.2021.3133100

Performance Evaluation of INT8 Quantized Inference on Mobile GPUs

SUMIN KIM, GUNJU PARK, AND YOUNGMIN YI
Department of Electrical and Computer Engineering, University of Seoul, Dongdaemun-gu, Seoul 02504, South Korea
Corresponding author: Youngmin Yi (ymyi@uos.ac.kr)

This work was supported by the Institute for Information & communications Technology Planning & Evaluation (IITP) grant funded by the Korean Government (MSIT) under Grant 2017-0-00142.

ABSTRACT During the past several years, the need for on-device deep learning has rapidly increased, and the performance of mobile GPUs has significantly increased. As a viable approach for efficient on-device deep learning, INT8 quantized inference has been actively studied and proposed but there are currently few frameworks that support INT8 quantization for mobile GPUs. This paper presents a unified framework that integrates various INT8 quantization methods, such as symmetric, asymmetric, per-layer, and per-channel, and discusses their impact on accuracy and efficiency on recent mobile GPUs. Moreover, we discuss the performance and accuracy of INT8 quantized Winograd convolution and propose INT8 Winograd convolution with F(2×2, 3×3), where weight tensors are quantized in INT4 and input tensors are quantized in INT6. We evaluated the performance of INT8 methods, including INT8 Winograd, for ResNet50, MobileNet-v1, and VGG16 on Mali G52, G72, and G76 GPUs on Odroid N2, Galaxy S9, and Galaxy Note 10+, respectively. INT8 quantized inference based on General Matrix Multiplication (GEMM) was 1.67× faster than FP32 GEMM for ResNet50 on Mali G52, and was further accelerated by batch normalization folding and by the proposed INT8 Winograd convolution, achieving 2.45× speedup in total with an accuracy loss of only 0.31%p.

INDEX TERMS On-device deep learning, INT8 quantization, INT8 Winograd convolution, mobile GPU.

I. INTRODUCTION

Deep learning has become a prevalent approach and demand for on-device deep learning in particular has rapidly increased in various domains such as autonomous driving, smart IoT, robots, and so forth. On-device deep learning offers several significant advantages compared with cloud based deep learning where the device connects to the cloud whenever it needs to process a deep learning task; (i) Users do not need to send private data to the cloud, and (ii) network communication is not required, and hence (iii) energy consumption and latency can be reduced.

However, mobile devices have significantly limited resources and power consumption constraints compared with cloud based systems. Recent GPUs in the cloud servers have over a hundred streaming multiprocessors, whereas mobile GPUs even in the current high-end smartphones have only approximately a dozen compute units. Thus, it is challenging to achieve real-time on-device deep learning inference, which is required for many applications. For example, autonomous vehicles and robots process object detection and recognition from incoming video streams in real-time.

Many studies have proposed various approaches to tackle this problem, including model compression to fit DNNs into smaller device memory [1], [2], hardware accelerators for deep neural networks (DNNs) and neural processing units (NPUs) [3]–[5], optimized convolution algorithms such as Winograd convolution [6], and lightweight models for mobile devices [7], [8]. Another approach is low-precision quantization, such as FP16 and INT8, to increase memory bandwidth and computation throughput, which is the theme of this paper.

NPUs have outperformed GPUs in terms of performance per Watt, due to which they have been rapidly adopted in mobile devices recently [3], [4]. However, they have their own limitations; it is difficult or impossible for NPUs to support custom layers since NPU APIs are not publicly available or have tightly restricted usage. In contrast, timely algorithm or model updates are possible with mobile GPUs. In addition, recent mobile GPUs incorporate hardware

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

acceleration for dot products that can complete a dot product with four-element vectors in a single cycle [9]. We envision that mobile GPUs will continue to be an essential component for deep learning in mobile devices, along with NPUs.

Therefore, this paper focuses on INT8 quantization for mobile GPUs. Although many on-device deep learning frameworks, including TensorFlow Lite, support INT8 quantization for CPUs, only a few frameworks currently support INT8 quantization for mobile GPUs: ARM Compute Library (ACL) since version v17.12 [10] and TensorRT. However, the former is not a high-level deep learning framework, and the latter is proprietary to NVIDIA and only works for Jetson GPUs that consume dozens of Watts. This lack is due to earlier mobile GPU performance being significantly inferior to mobile CPUs, but many recent mobile GPUs outperform their corresponding CPUs [4], [11] and hence INT8 quantization for mobile GPUs is crucial to achieve efficient on-device deep learning.

This paper presents a framework for INT8 quantized inference where various methods for INT8 quantization, such as symmetric, asymmetric, per-channel, and per-layer, are implemented. We discuss the INT8 quantization methods and evaluate their performance (i.e., efficiency) and accuracy impact on different mobile GPUs. Moreover, we discuss INT8 quantization for Winograd convolution, which is widely used for 3×3 convolution to reduce the number of multiplication. INT8 quantization cannot be directly applied to Winograd convolution because it requires transformation of tensors, requiring extra bits. We propose to use F(2×2, 3×3) rather than F(4×4, 3×3), with carefully selected bitwidth to maintain accuracy. The proposed INT8 Winograd convolution can utilize the same INT8 General Matrix Multiplication (GEMM) routine.

We implemented quantization methods in Caffe HRT [12] and evaluated performance and accuracy impact of the methods on widely used CNNs such as ResNet50 and MobileNet v1 on Mali-G52, which supports hardware acceleration of dot product. Considering the latency overhead and accuracy impact, symmetric with per-channel quantization was desirable for CNNs used in this work. Adding INT8 Winograd convolution and Batch Normalization folding, INT8 quantized convolution achieved 2.5–3.5× speedups compared to FP32 GEMM-based convolution with a negligible accuracy loss of 0.3–1.3%p.

Overall, this work makes the following contributions:

- Various INT8 quantization methods such as symmetric, asymmetric, per-channel, and per-layer were discussed and implemented in an open-source framework.
- INT8 quantization for Winograd convolution was discussed and proposed; To preserve accuracy, INT8 Winograd convolution should employ F(2×2, 3×3) with weights in INT4 and input feature maps in INT6.
- Extensive experiments for performance comparisons of different INT8 methods on recent mobile GPUs.

The remainder of this paper is organized as follows. Section II discusses related on-device deep learning and INT8 quantization studies, and Section III presents the background of INT8 quantization and the support in the recent library and GPUs. Section IV describes the proposed INT8 framework and discusses various INT8 methods. Section V discusses the challenges in INT8 quantized Winograd convolution and proposes INT8 Winograd convolution that can maintain accuracy. Section VI provides evaluation results and analysis for various INT8 methods, including the proposed INT8 Winograd convolution. Finally, Section VII summarizes and concludes the paper.

II. RELATED WORK

A. DEEP LEARNING ON MOBILE GPUs

Most current on-device deep learning frameworks use mobile CPUs rather than mobile GPUs [13]–[16], mainly because previous GPU performance was insufficient for efficient deep learning [5] and lightweight edge devices in IoT applications typically have very limited resources often not including GPUs [5]. Hence only a few works studied on-device deep learning utilizing mobile GPUs [17], [18]. However, modern GPUs have significantly increased processing cores and outperform mobile CPUs [3], [4]. For example, Lee *et al.* [11] showed that inference on the iPhone XS GPU was 5× faster than the CPU in the same SoC for MobileSSD, and 12× for MobileNet v1.

B. INT8 QUANTIZATION FOR DEEP LEARNING

INT8 quantization for deep learning has been attempted especially for mobile platforms as it can reduce model size [19] and computation time compared with FP32. Jacob *et al.* [20] introduced deep learning training with INT8 quantization for weights and activation maps and showed that many CNNs, including ResNet, Inception, and MobileNet, could benefit from INT8 quantization for the inference with negligible accuracy loss.

Several studies have considered converting DNNs trained initially with FP32 into INT8 models. Migacz [21] showed how to find a suitable scale factor and presented the results using *dp4a*, an instruction for efficient INT8 dot product computation on NVIDIA GPUs. Krishnamoorthi *et al.* [22] analyzed accuracy from various INT8 quantization representations and granularities, as well as end-to-end execution time, even if the results were obtained on DSPs, not on GPUs.

Kim *et al.* [23] proposed a co-operative approach where convolution layer filters were distributed over the CPU and GPU to execute the layer simultaneously. They only applied INT8 quantization for the CPU because GPU INT8 quantization exhibited lower efficiency than FP16 quantization, even on Mali T760, a high-end mobile GPU. However, contemporary mobile GPUs, such as Mali G52 and many others, offer significantly improved INT8 quantized inference support [3].

C. INT8 WINOGRAD CONVOLUTION

Winograd convolution is widely used for 3×3 convolution to reduce the number of multiplications. However, INT8

TABLE 1. INT8 support in on-device deep learning frameworks.

	TF Lite [27]	PyTorch [28] Mobile	Core ML [29]	TensorRT [30]
CPU INT8 support	O	O	O	O
GPU INT8 support	X	X	-	O
NPU INT8 support	Using NNAPI	X	Apple's NPU	Tensor Core
INT8 Winograd support	X	X	X	X

quantization cannot be directly applied to Winograd convolution; it requires transformation of tensors to the Winograd domain, but the large denominators in the transformation matrices would significantly degrade the accuracy if conventional INT8 quantization is applied. There are few works that studied INT8 quantization for Winograd convolution. Meng *et al.* [24] proposed a complex Winograd convolution algorithm for F(4 × 4, 3 × 3) where the construction field is extended from rationals to complex to reduce the denominator values, as well as filter scaling to reduce the bidwidth required for weights. However, they did not report execution times on actual platforms, nor the complex number computation overheads. In contrast, we propose a new INT8 quantized Winograd algorithm for F(2 × 2, 3 × 3) which carefully defines bitwidth smaller than INT8 for feature maps and weights to utilize the same INT8 GEMM routine after the transformation, and we evaluate the efficiency of the proposed method on mobile GPUs. Yao *et al.* [25] proposed Range-Scaled Quantization (RSQ) training method for INT8 Winograd algorithm, targeting one-dimensional convolution (Conv1D) equipped Automatic Speech Recognition (ASR) model inference on mobile CPUs. It does not discuss INT8 Winograd convolution for two-dimensional convolution (Conv2D) in image recognition, and it is quantization-aware training (QAT) approach as RSQ is applied *during* fine-tuning of the pre-trained model. Li *et al.* [26] proposed QAT algorithm called Lance for INT8 Winograd convolution for Conv2D. Our INT8 Winograd convolution is post training quantization (PTQ) approach (see Section III-B) and is for Conv2D.

D. ON-DEVICE DEEP LEARNING FRAMEWORK

On-device inference has only recently become commonly practical on mobile GPUs, and many on-device deep learning frameworks still do not support INT8 quantization for GPUs. Table 1 shows the current support for quantization in popular on-device deep learning frameworks. TensorFlow Lite [27], [31], a widely used on-device deep learning framework, and PyTorch Mobile [28], [32] only support INT8 quantization for CPUs using QNNPack [33]. Core ML [29] from Apple supports INT8 quantization, but it is unclear whether INT8 quantization for GPU is supported. In addition, it only supports weight quantization. TensorRT [30] supports INT8 quantization only for NVIDIA GPUs, and NVIDIA has yet to release low-power mobile GPUs for smartphones.

III. BACKGROUND

A. INT8 REPRESENTATION

INT8 representation quantizes inputs into 8-bit integers, requiring less storage space and typically faster than the original FP32 representation. INT8 is sufficient to maintain almost the same CNN model accuracy [22]. The input and weight tensors in a CNN are quantized for IN8 representation using the FP32 scale factor [20].

The original FP32 value, r, is linearly quantized into an INT8 value, q,

$$q = CLAMP\left(round\left(\frac{r}{S} + Z\right)\right), \tag{1}$$

where S is the FP32 scale factor, Z is the zero point. Each layer has unique S and Z values, and q is subsequently saturated to the limit if q exceeds the limit. Dequantization is computed in the opposite direction,

$$r = S(q - Z). \tag{2}$$

Scale factors and zero points are obtained for each layer before CNN inference. Weights are quantized statically since they do not change in the inference, whereas feature maps are quantized dynamically during inference. INT8 reduces tensor size four-fold, i.e., from 32 to 8 bit, also creating opportunities for faster inference (see Section IV-A). Bias is quantized into INT32 since it is added to the weight and feature map product, with scale factor also being the product of the two component scale factors.

B. SCALE FACTORS

It is crucial to find the most suitable scale factor and zero point to minimize quantization error. Scale factors can be obtained from PTQ or QAT approaches, where PTQ finds scale factors from trained model, and hence requires less time; whereas QAT finds scale factors by re-training [20] which requires more time but can minimize accuracy loss. Accuracy loss from PTQ depends on the specific method employed. For example, TensorRT uses KL-divergence [21], which we also employ in this work.

C. ARM COMPUTE LIBRARY

ARM Compute Library (ACL) is a software library for efficient DNN inference on ARM platforms [10], implemented using NEON for CPUs and OpenCL for GPUs. ACL also provides a high-level API, ACL Graph, that supports INT8 quantization for GPUs.

D. INT8 DOT PRODUCT ACCELERATION

ARM Mali Bifrost GPUs, such as G52 and G76, support hardware acceleration for INT8 dot product [9], [34]. An INT8 dot product for 4-element vectors is computed in in a single instructions (G52 and G72), or even in a single cycle (Mali G76), via built-in functions called `arm_dot` [35].

IV. INT8 QUANTIZED GEMM CONVOLUTION

This section proposes an INT8 quantization and inference framework for mobile GPUs, then it explains how INT8

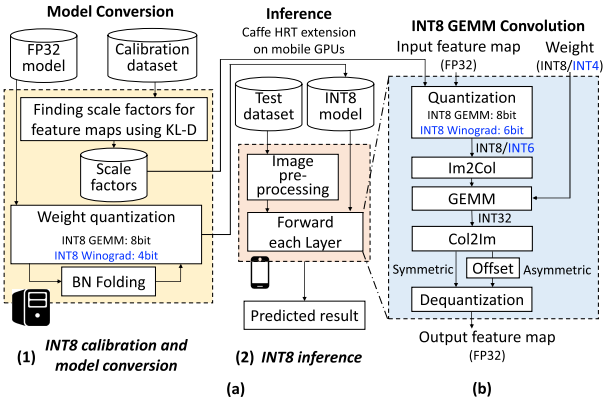

FIGURE 1. Proposed INT8 quantized CNN inference (a) framework and (b) INT8 GEMM convolution routine inside the framework.

quantization methods are implemented in the proposed framework.

A. PROPOSED FRAMEWORK OVERVIEW

As discussed in Section II, no current on-device deep learning framework except for some proprietary ones supports INT8 quantization for mobile GPUs. Figure 1(a) shows the INT8 quantization and inference flow of the proposed on-device deep learning framework. Figure 1 (a)(1) shows model conversion from FP32 to INT8, as well as finding scale factors for feature maps in advance, typically on a server. Figure 1 (a)(2) shows the quantized inference on a mobile GPU, where inference framework has been implemented by extending Caffe HRT. Figure 1 (b) shows the INT8 GEMM routine inside the framework.

As the proposed approach is PTQ, it finds scale factors for feature maps, which will be used during inference, using a calibration dataset. It is important that the scale factors do not degrade model performance. We obtain scale factors and zero points for each feature map in advance using KL-Divergence method [21]. It finds the scale factor that makes quantized value distribution, $Q(x)$, most similar to the original distribution, $P(x)$, when the KL-divergence is defined as

$$KL_{divergence}(P||Q) = -\sum_{i=a}^{b} p_i \log q_i + \sum_{i=r_{min}}^{r_{max}} p_i \log p_i, \quad (3)$$

where a and b are the quantized values from the thresholds, r_{min} and r_{max}, to ignore outliers in r, using Equation (1). It finds the scale factor that minimizes KL-divergence, varying r_{min} and r_{max} in Equation (3).

Weights are statically quantized before inference as they do not change during inference. They are easier to quantize than feature maps because their distribution is narrow and typically symmetric to the zero point [36]. Thus scale factors for weights are obtained simply using min and max values of FP32 weights and INT8 quantized weights are in $[-127, 127]$. The least value, -128, is excluded from the range to preserve the symmetry in weight distribution [37].

Once the scale factors for feature maps and the quantized weights, i.e., INT8 model, are obtained, the inference is made on a mobile GPU. As shown in Figure 1(b), the quantization is added at the start of the forward for each layer, then the dequantization is added at the end of the layer, implementing Equation (1) and (2), respectively. Most deep learning frameworks employ general matrix multiplication (GEMM) to perform convolution, rather than direct convolution; it first transforms the input image tiles to be convolved into columns of a matrix, which is referred to as image-to-column, or im2col. Then, convolution can be computed as a matrix multiplication of the transformed input matrix and the filter matrix, which can lead to better locality than the direct convolution in many cases. After the matrix multiplication, the output matrix is transformed back to the original shape, which is referred to as column-to-image, or col2im. In INT8 GEMM routines, GEMM outcomes are represented in INT32. Even if both inputs are INT8 and hence INT16 would be sufficient for the product, accumulating multiple products during dot product requires INT32 to prevent potential overflow (GEMMLowp [38]). The output is then dequantized into FP32, because the next layer could have a different scale factor or may not employ INT8 quantization.

A GEMM routine for a mobile GPU is typically implemented using vector instructions for Single Instruction Multiple Data (SIMD) rather than scalar instructions since most mobile GPUs support OpenCL vectors, which can significantly increase the efficiency. Thus, the proposed INT8 GEMM routines, as well as the im2col and col2im routines, use uchar8, i.e., a vector type for eight unsigned chars. Note that the GEMM routine in ACL uses float4 for the FP32 inputs and also uchar8 for the INT8 inputs.

The quantization and inference for INT8 Winograd convolution utilizes the same INT8 GEMM routine. However, as highlighted in a blue font in Figure 1, it requires different size of bits for both tensors compared to the GEMM-based convolution, which will be discussed in Section V.

B. INT8 QUANTIZATION AND OPTIMIZATION

1) SYMMETRIC AND ASYMMETRIC METHODS

When $Z = 0$ in INT8 representation, Equation (1) and (2) can be restated as

$$q = CLAMP\left(round\left(\frac{r}{S}\right)\right) \quad (4)$$

$$r = Sq, \quad (5)$$

called **symmetric** INT8, and hence $q \in [-128, 127]$. It is easier to compute convolutions with symmetric INT8 representation.

Conventional 2D convolution in FP32 can be expressed as

$$Y_{n,x,y} = \sum_{c,i,j} W_{n,c,i,j} \cdot X_{c,x+i,y+j}, \quad (6)$$

where X, W, and Y are input feature map, input feature weight, and output feature map for the convolution layer. Therefore, Equation (6) can be approximated using INT8

quantization as

$$\begin{aligned} Y_{n,x,y} &\approx (S^W Q_n^W) * (S^X Q_{x,y}^X) \\ &= S^W \cdot S^X \cdot (Q_n^W * Q_{x,y}^X) \\ &= S^W \cdot S^X \cdot \sum_{c,i,j} Q_{n,c,i,j}^W \cdot Q_{c,x+i,y+j}^X \cdot \end{aligned} \tag{7}$$

Thus, INT8 convolution is first computed similar to FP32, then only needs to multiply the scale factors.

The process is called **asymmetric** INT8 if $Z \neq 0$, and quantization proceeds as Equation (1) and convolution as

$$\begin{aligned} Y_{n,x,y} &\approx S^W \cdot S^X \cdot ((Q_n^W + Z^W) * (Q_{x,y}^X + Z^X)) \\ &= S^W \cdot S^X \cdot \Big(\sum_{c,i,j} Q_{n,c,i,j}^W \cdot Q_{c,x+i,y+j}^X \\ &\quad + \sum_{c,i,j} Z^X Q_{n,c,i,j}^W + \sum_{c,i,j} Z^W Q_{c,x+i,y+j}^X \\ &\quad + \sum_{c,i,j} Z^W Z^X \Big). \end{aligned} \tag{8}$$

In contrast to Equation (7), asymmetric INT8 convolution requires additional computation since $Z \neq 0$. We use offset computation for efficient computation, i.e., $Z^{IN} \Sigma Q^W$, $Z^W \Sigma Q^{IN}$, and $\Sigma Z^W Z^{IN}$ are calculated separately and added to $\Sigma Q^W \cdot Q^{IN}$. We can calculate $Z^{IN} \Sigma Q^W$ and $\Sigma Z^W Z^{IN}$ prior to inference since they do not have input feature maps.

The process shown in Figure 1 supports both symmetric and asymmetric INT8. Offset computation should be added after GEMMLowp to implement asymmetric INT8 and quantization should be modified to consider the zero point. Note that ACL provides only INT8 GEMM routine for the asymmetric INT8 convolution; neither routines for symmetric INT8 convolution nor the scale factor calculation modules.

Scale factors for both methods are derived as follows: Symmetric INT8 quantization first makes a histogram for positive input feature map values with 127 bins, and then finds the threshold to minimize KLD (discussed in Section III-B). Weight quantization should quantize maximum absolute weights onto ± 127. In contrast, asymmetric quantization requires the specific zero point. Therefore it finds two thresholds, max and min, for an input feature map, from which a scale factor and a zero point are obtained.

2) PER-CHANNEL AND PER-LAYER METHODS

A layer can have multiple scale factors and zero points, each channel in the layer having its own scale factor and zero point, which is called **per-channel** rather than **per-layer** quantization [22]. Per-channel quantization can improve approximation and accuracy because it uses more suitable scale factors differently for each channel in the layer. However, this requires more memory accesses to scale factor and zero point arrays during quantization and dequantization.

3) BATCH NORMALIZATION FOLDING

The batch normalization layer normalizes input feature maps for mini-batches by recentering and rescaling,

$$OUT = \gamma \cdot \frac{IN - \mu}{\sigma} + \beta, \tag{9}$$

which improves network stability, and hence network performance [39].

Batch normalization (BN) folding integrates BN layer parameters into the convolution layer, where convolution layer weights, W, and bias, B, are modified into W_{folded} and B_{folded} as

$$\begin{aligned} W_{folded} &= \frac{\gamma}{\sigma} W, \\ B_{foled} &= \frac{\gamma}{\sigma} (B - \mu) + \beta. \end{aligned} \tag{10}$$

BN folding can accelerate inference by reducing memory accesses by fusing BN and convolution layers. There are only a few arithmetic operations per element in BN, without any dot product in particular. Thus, INT8 quantized computations are unnecessary considering their computational overhead, and we apply INT8 quantization after BN folding instead. For W_{folded}, we need not modify quantized weights but only multiply the original weight scale factor by $\frac{\gamma}{\sigma}$ to update the scale factor for symmetric INT8 quantization. Unlike the original convolution layers, we can no longer neglect the bias but must apply appropriate quantization for the bias in BN folded convolution layers. For B_{folded}, the scale factor of bias is updated by multiplying the weight scale factor and feature map scale factor, and hence it is quantized to INT32.

V. INT8 WINOGRAD CONVOLUTION

This section starts with the introduction of Winograd convolution algorithm, then it discusses the accuracy impact due to INT8 quantization and proposes suitable INT8 quantization for Winograd convolution that does not degrade accuracy.

A. FP32 WINOGRAD

Winograd convolution reduces the number of multiplications at the cost of increased additions [6], [40] and can be applied to 2D convolution with filter size 3×3 and 5×5 when stride $= 1$ [41]. Thus it is widely used for 3×3 convolution, which is the most popular 2D convolution, as it typically leads to better efficiency than GEMM-based convolution. As shown in Equation (11), Winograd convolution transforms the input and weight tensors into the Winograd domain where the number of multiplications is reduced to *the minimal filters* and the multiplications are subsequently performed in a batched GEMM. Then, the output tensor in the Winograd domain is transformed back to the original domain.

$$r_{out} = A^T \Big(GwG^T \odot B^T r_{in} B \Big) A, \tag{11}$$

where w is a filter, r_{in} is an input tile, r_{out} is an output tile in the original domain, and $\odot$ is an element-wise product. B, G, and A are transformation matrices for input(r_{in}), weight(w), and

output in the Winograd domain, respectively. Output tile size differs depending on filter size: it can be 4×4 and 2×2 for 3×3 filer size, which is the most widely employed filter size in 2D convolution. In general, Winograd convolution with filter size r×r and output tile size m×m is denoted as F(m×m, r×r). Transformation matrices for F(2 × 2, 3 × 3) introduced in [6] are as follows:

$$B^T = \begin{bmatrix} 1 & 0 & -1 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & -1 & 1 & 0 \\ 0 & 1 & 0 & -1 \end{bmatrix},$$
$$G = \begin{bmatrix} 1 & 0 & 0 \\ \frac{1}{2} & \frac{1}{2} & \frac{1}{2} \\ \frac{1}{2} & -\frac{1}{2} & \frac{1}{2} \\ 0 & 0 & 1 \end{bmatrix},$$
$$A^T = \begin{bmatrix} 1 & 1 & 1 & 0 \\ 0 & 1 & -1 & -1 \end{bmatrix}. \quad (12)$$

Since B is a constant matrix of integers, the transformation can be done as additions, and other transformations can also be done using additions. Element-wise multiplications still require multiplications but the number of multiplications is smaller than matrix multiplication. Element-wise multiplication can be efficiently computed as a batched GEMM by reshaping the matrices.

As the number of multiplications is larger in F(4×4, 3×3) than in F(2 × 2, 3 × 3), Winograd convolution achieves 4× reduction in multiplication operations for F(4 × 4, 3 × 3), whereas 2.25× reduction for F(2×2, 3×3). In contrast, since the denominator in the G matrix is larger in F(4 × 4, 3 × 3) than F(2 × 2, 3 × 3), convolution computation errors using the former tend to be larger than the latter [42]. On the other hand, Winograd convolution requires more memory space to store transformed results [43].

B. INT8 WINOGRAD F(4 × 4, 3 × 3)

No current deep learning framework supports INT8 Winograd convolution. This subsection discusses the challenges and issues in INT8 Winograd convolution.

In conventional Winograd convolution in FP32, F(4 × 4, 3 × 3) can achieve larger speedup than F(2 × 2, 3 × 3), as discussed in Section V-A. However, F(4 × 4, 3 × 3) is not suitable for INT8 quantized computation. The weight transformation matrix for F(4 × 4, 3 × 3) is

$$G = \begin{bmatrix} \frac{1}{4} & 0 & 0 \\ -\frac{1}{6} & -\frac{1}{6} & -\frac{1}{6} \\ -\frac{1}{6} & \frac{1}{6} & -\frac{1}{6} \\ \frac{1}{24} & \frac{1}{12} & \frac{1}{6} \\ \frac{1}{24} & -\frac{1}{12} & \frac{1}{6} \\ 0 & 0 & 1 \end{bmatrix}, \quad (13)$$

which includes division by 24 and hence is unsuitable for INT8 representation since this would increase quantization error. To avoid division, one could first multiply each matrix, G and G^T, by 24 in advance, and then divide the transformed output in INT32 by $24^2 = 576$. However, multiplication of both G and G^T by 24 would multiply some elements in w by $24^2 = 576$. Since the maximum value an INT8 can represent even in an unsigned form is $2^8 - 1 = 255$, we cannot apply this approach and F(4 × 4, 3 × 3) cannot retain accuracy with INT8 GEMM.

Let us consider Winograd convolution in F(4 × 4, 3 × 3) could be done in INT16 GEMM. Input and weight are in INT8, and multiplying the constant in advance requires more bits; filter transformation needs to multiply 576 and input transformation 100 in the worst case, requiring additional 10 and 7 bits, respectively. Thus, transforming from INT8 matrices would require 18 and 15 bits for weights and inputs, respectively. However, if weights were quantized in 6 bits rather than 8, transformation with constant multiplication could fit in INT16 and GEMM could be done in INT16. Although INT16 GEMM can be slower than INT8 GEMM, the efficiency loss is relatively small. On the other hand, if we truncate 8 bits to make the results fit in INT8 by shifting INT16 results to the right by 8 bits, error becomes too large. Therefore, F(4 × 4, 3 × 3) is not suitable in INT8.

C. INT8 WINOGRAD F(2 × 2, 3 × 3)

INT8 quantization for F(2 × 2, 3 × 3), on the contrary, requires only additional 4 bits for weight and 2 bits for input; multiplication by 9 and 4 are required in the weight transformation using G and G^T in Equation (12), and in the input transformation using B and B^T, respectively. If we lower precision by quantizing weight and input to INT4 and INT6, then both results can be represented in INT8. Error due to lowered precision (i.e., 4 and 2 bits for weight and input, respectively) remains negligible (see Section VI).

Therefore, we propose INT8 Winograd convolution F(2 × 2, 3 × 3) using INT8 GEMM, as shown in Figure 2(b). FP32 inputs are quantized in INT6 to allow input transformation to fit in INT8, and FP32 weights are statically quantized in INT4 to ensure that filter transformation results can also fit in INT8. Each transformation matrix in the filter transformation, G and G^T, is multiplied by 2 in advance to avoid divisions. Thus, batched GEMM output is INT32 and should be divided by $2^2 = 4$. Note that element-wise multiplications are computed as a batched GEMM as mentioned before.

VI. EXPERIMENTS

A. SETUP

We used Caffe HRT [12] as the baseline deep learning framework for our experiments, and extended it to support the proposed INT8 GEMM and INT8 Winograd convolution quantization. Caffe HRT is built on top of ACL (v19.05) and provides the same Caffe interface, which allowed us to easily implement the discussed quantization methods. We implemented in Caffe HRT the INT8 features that ACL does not

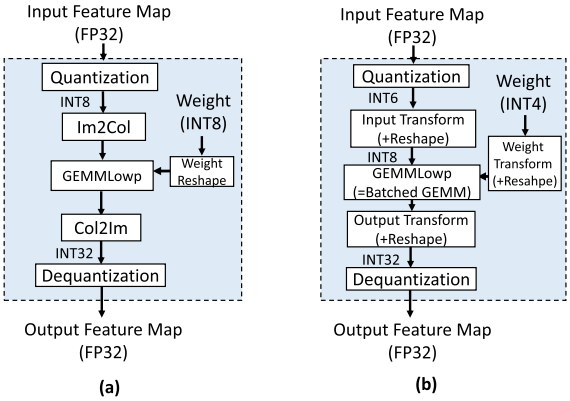

FIGURE 2. (a)INT8 GEMM convolution and (b)INT8 Winograd F(2 × 2,3 × 3) convolution with INT6 and INT4; it can employ the same INT8 GEMM routine.

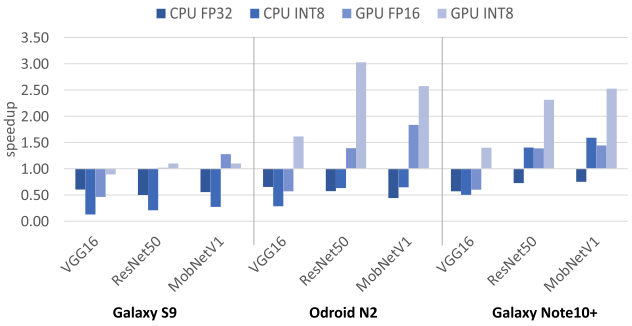

FIGURE 3. CNN inference speedup on various devices with FP32, FP16, and INT8 representations. Baseline is FP32 GPU for each device.

support, such as INT8 Winograd convolution, symmetric methods, and per-channel method; and extended Caffe HRT to utilize INT8 features ACL supports, such as asymmetric and per-layer method. Since ACL has neither scale factor calculation, nor BN folding, those features were implemented in the proposed framework.

CNN models used in the evaluation were ResNet50, VGG-16, and MobileNet v1. FP32 models were pre-trained Caffe models, from which INT8 models were obtained in the proposed framework.

Devices used in the experiment were Galaxy S9 with Mali G72 MP18 [44], [45], Note 10+ with Mali G76 MP12 [46], [47], and Odroid N2 with Mali G52 MP2 [48], which all supported INT8 dot product hardware acceleration (see Section III-D). ACL Graph was run on the three devices whereas Caffe HRT was only run on Odroid since it cannot run on Android.

B. ON-DEVICE INFERENCE ON CPUs AND GPUs

Before examining the effect of the proposed INT8 methods including INT8 Winograd convolution, we compared INT8 quantized CNN inference with FP16 and FP32 inference on various mobile GPUs to confirm that INT8 quantized inference was the fastest. Although Caffe HRT provides the same Caffe interface and employs ACL low-level APIs, data copy between Caffe BLOB and ACL tensor makes it unsuitable for measuring and comparing end-to-end execution time of CNNs. Therefore, we measured execution time with ACL Graph (v19.05), which supports INT8 quantized CNN inference for both CPUs and GPUs. Note that ACL Graph supports neither INT8 Winograd convolution nor any other INT8 methods discussed in this paper. Consequently, FP32 and FP16 convolution embraced GEMM and Winograd implementations, whereas INT8 convolution was only implemented with GEMM.

Figure 3 shows the speedups of INT8 and FP16 quantized inference on a CPU and a GPU, compared to FP32 inference on a GPU. On Odroid N2 and Galaxy Note 10+, INT8 GPU is the fastest with all three CNN models, showing large speedups against FP32 GPU; up to 3.03× with ResNet50 on N2, and up to 2.5× with MobileNet v1 on Note 10+, which corresponds to 1.6× faster than INT8 CPU on Note 10+. INT8 CPU, as well as INT8 GPU, is faster than FP32 GPU in most cases on Note 10+ since it is equipped with Cortex A75, which is much faster than Exynos M3 in S9 and Cortex-A73 in N2.

However, on Galaxy S9, INT8 GPU is not the fastest; FP32 GPU is the fastest with VGG16, and FP16 GPU is fastest with MobileNet v1. INT8 GPU is fastest only for ResNet50, but by less than 10%. This is because Mali G72 in S9 does not support single-cycle integer dot product acceleration unlike Mali G76 in Note 10+. Mali G52 in N2 neither supports it but, as the cache size in Mail G52 is smaller than Mali G72, the locality improvement due to INT8 quantized inference is relatively larger in Mali G52 than in Mali G72 in S9. Due to the similar reason, INT8 CPU is slower than FP32 CPU in S9; Exynos M3 in S9 and Cortex-A73 in N2 are based on ARMv8 Instruction Set Architecture (ISA) and do not support integer dot product extension; it is supported since ARMv8.2 ISA, which Cortex-A75 in Note10+ is based on. Also, the cache size of Cortex-A73 is smaller than that of Exynos M3. On the other hand, VGG16 results in smaller speedups than other models in all three devices. This is because VGG16 convolution layers are all 3 × 3 and are implemented in Winograd convolution in FP32. However, as ACL does not support INT8 Winograd convolution, INT8 quantized inference runs 3 × 3 layers in VGG16 as GEMM-based convolution, resulting in a smaller speedup or even a slower execution than FP32. Another reason is that the size of feature maps in VGG16 is larger, incurring larger overhead in quantization and dequantization.

In summary, INT8 inference is faster than FP16 or FP32 inference, and recent mobile GPUs lead to faster inference than CPUs.

C. INT8 METHODS FOR ON-DEVICE GPU INFERENCE

1) INT8 QUANTIZATION

This section examines effects of the proposed INT8 methods, including INT8 Winograd convolution, for efficiency and accuracy. Accuracy and execution time were measured in Caffe HRT rather than ACL Graph, on Odroid N2 (Mali G52),

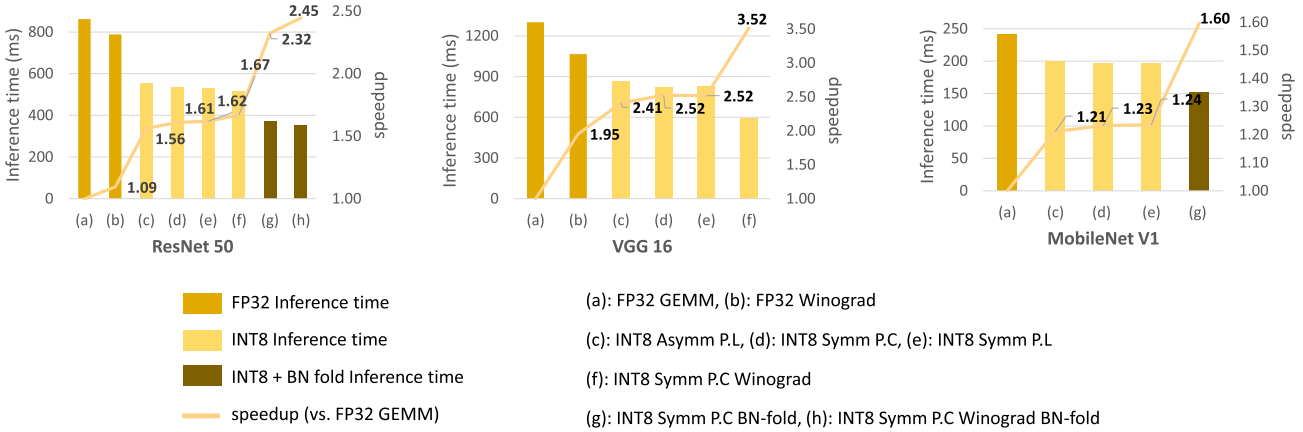

FIGURE 4. End-to-end execution time for INT8 methods with ResNet50 on Mali G52 MP2. P.L: per layer, P.C: per channel.

TABLE 2. Top-1 accuracy of various INT8 methods for ImageNet.

Method	FP32	INT8					
		ASY	SYM				
		PL	PL	PC	B	W	B,W
ResNet50	74.1	73.6	73.8	74.0	74.1	72.5	73.8
VGG 16	69.7	69.2	69.4	69.5	-	68.5	-
MobNet v1	70.0	68.3	69.0	69.4	69.7	-	-

ASY: asymmetric, SYM: symmetric, PL: per layer, PC: per channel
B: BN folding(PC) W: Winograd(PC) MobNet: MobileNet

since we implemented each method in Caffe HRT, as shown in Table 2.

Accuracy loss is very small, generally less than −1.0%p except for asymmetric and per-layer (−1.7%p) with MobileNet v1, and INT8 Winograd convolution without BN folding; INT8 Winograd convolution showed −1.2%p for VGG16, which has no BN layers. With BN folded, however, INT8 Winograd accuracy loss was small for ResNet50 (−0.31%p). Winograd convolution cannot be applied for MobileNet v1 since it has no 3 × 3 convolution layers.

Accuracy difference between asymmetric and symmetric methods was small (0.1–0.7%p). However, asymmetric overhead was not negligible compared with symmetric (Figure 4). Therefore, we used symmetric method to measure effects of the other methods.

Per-channel is more accurate than per-layer method, as expected; although the differences are as small as 0.09%p for VGG16, accuracy from per-channel method with ResNet50 is also higher (0.21%p) than per-layer method. BN folding even increases accuracy; +0.13%p for ResNet50 and +0.3%p for MobileNet v1, compared to per-channel without BN folding.

Figure 4 shows an end-to-end execution time with the INT8 methods, including INT8 Winograd convolution. FP32 GEMM is the baseline but FP32 Winograd is also presented for comparison with INT8 Winograd because the speedup of INT8 Winograd over FP32 GEMM comes from INT8 quantization and Winograd algorithm itself, among which only the former is of our interest. Overall speedups with fastest INT8 quantized inference are 2.45× for ResNet50, 3.52× for VGG16, and 1.60× for MobileNet v1.

INT8 GEMM inference with asymmetric (Figure 4(c)), which turned out to be the slowest INT8 configuration, is still 1.21× to 2.41× faster than FP32 GEMM inference. BN folding enabled further speedups (compare Figure 4(g) and (h) with (d) and (f), respectively). BN folding integrates BN into the convolution layer, reducing computational load and global memory (see Section IV-A), providing a further 1.30–1.47× speedups.

Since Winograd convolution can only be applied to 3 × 3 and not 1 × 1 convolution layers, INT8 Winograd convolution speedups are presented only for ResNet50 and VGG16. In Figure 4 for ResNet50, INT8 Winograd convolution speedup over INT8 GEMM is not obvious as it is an end-to-end execution time including all layers, not just 3 × 3 layers. Thus Figure 4(f) for ResNet50 is similar to (d), and (h) is similar to (g). In contrast, as the convolution layers of VGG16 are all 3 × 3, inference time with INT8 Winograd convolution(Figure 4(f) for VGG16) is significantly smaller than INT8 GEMM inference time(Figure 4(d)).

INT8 symmetric method (Figure 4(e)) is 3–11%p faster than INT8 asymmetric method (Figure 4(c)) because asymmetric requires additional computation for the offset. Thus, symmetric method is preferable considering the negligible accuracy difference with asymmetric method as discussed before.

Per-channel (Figure 4(d)) is comparable or only slightly slower than per-layer. Therefore, we used per-channel when measuring other methods since accuracy loss for per-channel is smaller than per-layer.

2) INT8 GEMM LAYER ANALYSIS

Figure 5 shows speedups of INT8 GEMM over FP32 GEMM for each convolution layer. INT8 GEMM was computed with symmetric and per-channel methods without BN folding on Odroid N2. Speedups for 3 × 3 convolution layers are larger

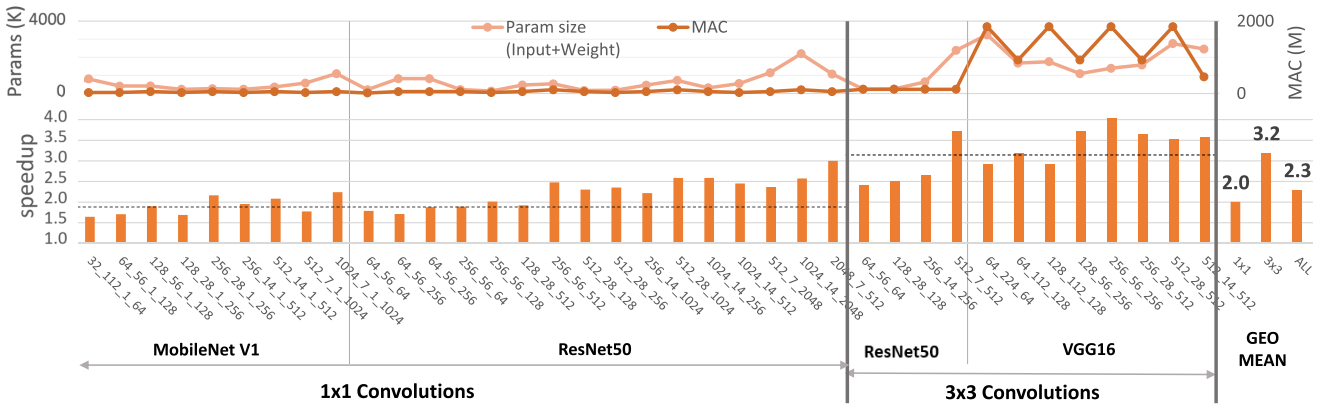

FIGURE 5. Layer-by-layer speedup of INT8 GEMM over FP32 GEMM(C_{in}-WH_{in}-C_{out}).

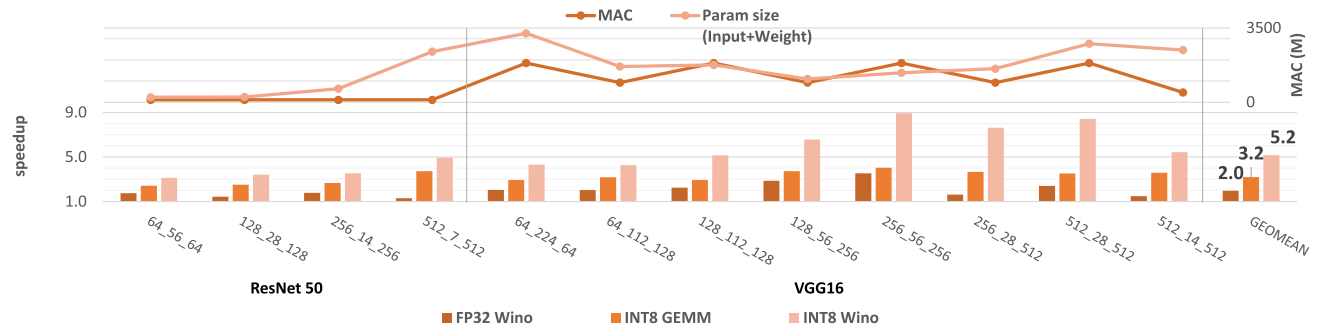

FIGURE 6. INT8 Winograd convolution speedup over FP32 GEMM for different shapes of layers (C_{in}-WH_{in}-C_{out}).

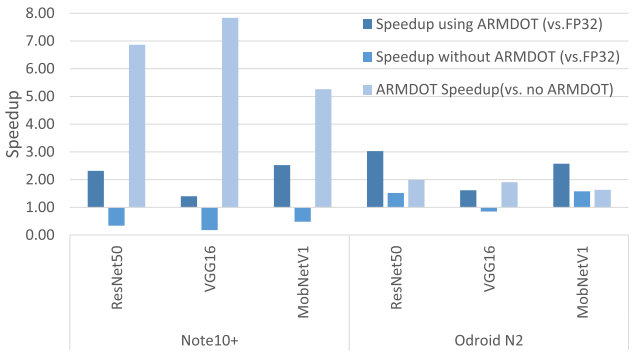

FIGURE 7. INT8 Inference speedup with `arm_dot`.

than for 1 × 1 convolution layers, with geometric mean speedup for 3 × 3 convolution being 3.2×, whereas 1 × 1 convolution 2.0×.

Speedups for 1 × 1 convolutions are largely proportional to parameter size; whereas speedup is not proportional to parameter size for VGG16, beyond certain amount of computation. Speedup from INT8 quantized convolution arises from two folds; (1) input and weight tensor sizes are 4× smaller, hence they can better fit in the cache, and (2) INT8 dot product for four-element vectors can be executed in a single instruction in a Mali GPU. When ALU operations are not the performance bottleneck, as in many current 1 × 1 convolution layers, speedup is affected by the first factor, i.e., increased INT8 GEMM locality. On the contrary, if ALU operations are the performance bottleneck, then those layers benefit from the second factor.

3) INT8 WINOGRAD LAYER ANALYSIS

Figure 6 shows 3 × 3 convolution layer speedups of INT8 Winograd convolution over FP32 GEMM. Again, Winograd convolution is not applicable for MobileNet v1 as it does not have 3 × 3 convolution layers. INT8 GEMM is faster than FP32 Winograd convolution for all layers, and INT8 Winograd convolution is faster than INT8 GEMM. Mean speedup from INT8 Winograd convolution is 5.2×, which corresponds to 2.6× compared to FP32 Winograd convolution and 1.63× compared to INT8 GEMM convolution. As Winograd convolution reduces the number of multiplications (Section V), it significantly improves performance when the bottleneck is ALU operations; most VGG16 layers achieves more than 2× speedups.

4) INT8 DOT PRODUCT HARDWARE ACCELERATION IN GPU

Hardware acceleration for INT8 dot product is supported in GPUs since Mali G52 (see Section III) via a built-in function, `arm_dot`, which can be called inside an OpenCL kernel. The first and the second bar of Figure 7 are INT8 GEMM inference speedups against FP32 GEMM inference, with and

without arm_dot. The third bar shows the speedup of INT8 GEMM with arm_dot compared to without arm_dot. On Galaxy Note 10+ with Mali G76, arm_dot is executed in a single cycle, and hence arm_dot speedups are very large (5× to 8×). On Odroid N2 with Mali G52, however, arm_dot is executed in single instruction but not in a single cycle, with arm_dot speedups being in the range of 1.8× and 2.2×. Note that, on Mali G76, INT8 GEMM without arm_dot is even slower than FP32 GEMM. This is because the cache size in G76 is larger, reducing the relative gain from INT8 quantized model size.

VII. CONCLUSION

This paper studied INT8 quantized CNN inference accuracy and efficiency effects on mobile GPUs. We discussed different INT8 quantization methods, many of which are not supported in current on-device deep learning frameworks, such as ACL and TensorFlow Lite. We implemented them in a unified framework to compare performance. Moreover, we proposed INT8 Winograd convolution with F(2 × 2, 3 × 3) where weight tensors are quantized in INT4 and input tensors in INT6, considering extra bits required for the transformation. Consequently, the proposed INT8 Winograd can be computed in INT8 with negligible accuracy loss.

INT8 GEMM was 1.67× faster than FP32 GEMM for ResNet50 on Mali G52. With BN folding and INT8 Winograd convolution, it was further accelerated to 2.45×, while retaining accuracy (−0.31%p). The large speedups of INT8 quantized inference were possible due to hardware acceleration called arm_dot in Mali GPUs.

As the future work, we plan to evaluate the efficiency and accuracy of INT8 quantized inference on NPUs with various CNNs.

REFERENCES

[1] Y. He, J. Lin, Z. Liu, H. Wang, L.-J. Li, and S. Han, "AMC: AutoML for model compression and acceleration on mobile devices," in *Proc. Eur. Conf. Comput. Vis. (ECCV)*, Sep. 2018, pp. 784–800.

[2] Y. Cheng, D. Wang, P. Zhou, and T. Zhang, "A survey of model compression and acceleration for deep neural networks," 2017, *arXiv:1710.09282*.

[3] A. Ignatov, R. Timofte, A. Kulik, S. Yang, K. Wang, F. Baum, M. Wu, L. Xu, and L. Van Gool, "AI benchmark: All about deep learning on smartphones in 2019," 2019, *arXiv:1910.06663*.

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

[5] C.-J. Wu, D. Brooks, K. Chen, D. Chen, S. Choudhury, M. Dukhan, K. Hazelwood, E. Isaac, Y. Jia, and B. Jia, "Machine learning at facebook: Understanding inference at the edge," in *Proc. IEEE Int. Symp. High Perform. Comput. Archit. (HPCA)*, Feb. 2019, pp. 331–344.

[6] A. Lavin and S. Gray, "Fast algorithms for convolutional neural networks," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit. (CVPR)*, Jun. 2016, pp. 4013–4021.

[7] 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.

[8] 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*.

[9] ARM. (2018). *ARM Announces Mali-G76 GPU: Scaling up bifrost*. [Online]. Available: https://www.anandtech.com/show/12834/arm-announces-the-mali-g76-scaling-up-bifrost/3

[10] ARM. (2017). *Arm Compute Library*. [Online]. Available: https://developer.arm.com/technologies/compute-library

[11] J. Lee, N. Chirkov, E. Ignasheva, Y. Pisarchyk, M. Shieh, F. Riccardi, R. Sarokin, A. Kulik, and M. Grundmann, "On-device neural net inference with mobile GPUs," 2019, *arXiv:1907.01989*.

[12] OAID AI Lab. (2018). *Caffehrt*. [Online]. Available: https://github.com/OAID/Caffe-HRT

[13] 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.

[14] Y. Liu, Y. Wang, R. Yu, M. Li, V. Sharma, and Y. Wang, "Optimizing CNN model inference on CPUs," in *Proc. USENIX Annu. Tech. Conf. (USENIX ATC)*, 2019, pp. 1025–1040.

[15] L. Tang, Y. Wang, T. L. Willke, and K. Li, "Scheduling computation graphs of deep learning models on manycore CPUs," 2018, *arXiv:1807.09667*.

[16] A. Zlateski, K. Lee, and H. S. Seung, "ZNN–A fast and scalable algorithm for training 3D convolutional networks on multi-core and many-core shared memory machines," in *Proc. IEEE Int. Parallel Distrib. Process. Symp. (IPDPS)*, May 2016, pp. 801–811.

[17] 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.

[18] L. N. Huynh, Y. Lee, and R. K. Balan, "DeepMon: Mobile GPU-based deep learning framework for continuous vision applications," in *Proc. 15th Annu. Int. Conf. Mobile Syst., Appl., Services*, 2017, pp. 82–95.

[19] Google. *Tensorflow Model Optimization–Quantization*. Accessed: 2019. [Online]. Available: https://www.tensorflow.org/lite/performance/model_optimization

[20] 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.

[21] NVIDIA. (2017). *8-Bit Inference With Tensorrt*. [Online]. Available: http://on-demand.gputechconf.com/gtc/2017/presentation/s7310-8-bit-inference-with-tensorrt.pdf

[22] R. Krishnamoorthi, "Quantizing deep convolutional networks for efficient inference: A whitepaper," 2018, *arXiv:1806.08342*.

[23] 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.*, 2019, pp. 1–15.

[24] L. Meng and J. Brothers, "Efficient Winograd convolution via integer arithmetic," 2019, *arXiv:1901.01965*.

[25] Y. Yao, Y. Li, C. Wang, T. Yu, H. Chen, X. Jiang, J. Yang, J. Huang, W. Lin, H. Shu, and C. Lv, "INT8 Winograd acceleration for Conv1D equipped ASR models deployed on mobile devices," 2020, *arXiv:2010.14841*.

[26] G. Li, L. Liu, X. Wang, X. Ma, and X. Feng, "Lance: Efficient low-precision quantized Winograd convolution for neural networks based on graphics processing units," in *Proc. IEEE Int. Conf. Acoust., Speech Signal Process. (ICASSP)*, May 2020, pp. 3842–3846.

[27] Google. (2018). *Tensorflow Lite*. [Online]. Available: https://www.tensorflow.org/lite/

[28] Facebook. (2020). *Pytorchmobile*. [Online]. Available: https://pytorch.org/mobile/home/

[29] Apple. (2019). *Coreml*. [Online]. Available: https://developer.apple.com/documentation/coreml

[30] NVIDIA. (2017). *Tensorrt*. [Online]. Available: https://developer.nvidia.com/tensorrt

[31] M. Abadi. (2015). *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. [Online]. Available: http://tensorflow.org/

[32] A. Paszke *et al.*, "Pytorch: An imperative style, high-performance deep learning library," in *Proc. Adv. Neural Inf. Process. Syst.*, vol. 32, 2019, pp. 8026–8037.

[33] Facebook. (2018). *Qnnpack*. [Online]. Available: https://github.com/pytorch/QNNPACK

[34] ARM. (2018). *Mali-G76: Taking High-End Graphics to the Next Level*. [Online]. Available: https://community.arm.com/developer/tools-software/graphics/b/blog/post s/mali-g76-taking-high-end-graphics-to-the-next-level

[35] Khronos. (2019). *ARM Integer Dot Product*. [Online]. Available: https://www.khronos.org/registry/OpenCL/extensions/arm/cl_arm_integer_dot_product.txt
[36] M. Thoma, "Analysis and optimization of convolutional neural network architectures," 2017, *arXiv:1707.09725*.
[37] NVIDIA. (2017). *Low Precision Inference on GPU*. [Online]. Available: https://developer.download.nvidia.com/video/gputechconf/gtc/2019/presentation/s9659-inference-at-reduced-precision-on-gpus.pdf
[38] Google. *Gemmlowp*. Accessed: 2019. [Online]. Available: https://github.com/google/gemmlowp
[39] S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," in *Proc. Int. Conf. Mach. Learn.*, 2015, pp. 448–456.
[40] T. Kouya, "Accelerated multiple precision matrix multiplication using Strassen's algorithm and Winograd's variant," *JSIAM Lett.*, vol. 6, pp. 81–84, Mar. 2014.
[41] R. Xu, S. Ma, W. Li, and Y. Guo, "Accelerating CNNs using optimized scheduling strategy," in *Proc. Int. Conf. Algorithms Archit. Parallel Process.* Cham, Switzerland: Springer, 2018, pp. 196–208.
[42] K. Vincent, K. Stephano, M. Frumkin, B. Ginsburg, and J. Demouth, "On improving the numerical stability of Winograd convolutions," in *Proc. Int. Conf. Learn. Represent. Workshop*, 2017.
[43] A. Xygkis, L. Papadopoulos, D. Moloney, D. Soudris, and S. Yous, "Efficient winograd-based convolution kernel implementation on edge devices," in *Proc. 55th Annu. Design Autom. Conf.*, Jun. 2018, pp. 1–6.
[44] A. Frumusanu. (2018). *The Samsung Galaxy S9 and S9+ Review*. [Online]. Available: https://www.anandtech.com/show/12520/the-galaxy-s9-review/
[45] Samsung. (2018). *Samsung Galaxy S9 and S9+*. [Online]. Available: https://www.samsung.com/sec/business/smartphones/galaxy-s9/
[46] Samsung Electronics. (2019). *Galaxy Note10+*. [Online]. Available: https://www.samsung.com/global/galaxy/galaxy-note10/
[47] A. Frumusanu. (2019). *Galaxy Note10+–Full Phone Specifications*. [Online]. Available: https://www.gsmarena.com/samsung_galaxy_note10+-9732.php
[48] Hardkernel. (2019). *Odroid N2*. [Online]. Available: https://www.hardkernel.com/ko/shop/odroid-n2-with-4gbyte-ram/

SUMIN KIM received the B.S. degree in electrical and computer engineering from the University of Seoul, South Korea, in 2020, where he is currently pursuing the M.S. degree in electrical and computer engineering. His research interests include on-device deep learning utilizing mobile GPUs and hardware design for NPUs.

GUNJU PARK received the B.S. and M.S. degrees in electrical and computer engineering from the University of Seoul, South Korea, in 2019 and 2021, respectively. His research interest includes on-device deep learning.

YOUNGMIN YI received the B.S. and Ph.D. degrees in computer engineering from Seoul National University, in 2000 and 2007, respectively. He was a Postdoctoral Researcher with the University of California at Berkeley, Berkeley, USA, from 2007 to 2009, and a Senior Researcher with the Samsung Advanced Institute of Technology, from 2009 to 2010, before joining the School of Electrical and Computer Engineering, University of Seoul, where he is currently a Professor. His research interests include on-device deep learning, algorithm/architecture co-design for heterogeneous manycore platforms, and GPU computing.