PhoneBit: Efficient GPU-Accelerated Binary Neural Network Inference Engine for Mobile Phones

Gang Chen*, Shengyu He†‡, Haitao Meng†, Kai Huang*
*Sun Yat-sen University, China
†Notheastern University, China
‡ Peng Cheng Laboratory, China

***Abstract*—Over the last years, a great success of deep neural networks (DNNs) has been witnessed in computer vision and other fields. However, performance and power constraints make it still challenging to deploy DNNs on mobile devices due to their high computational complexity. Binary neural networks (BNNs) have been demonstrated as a promising solution to achieve this goal by using bit-wise operations to replace most arithmetic operations. Currently, existing GPU-accelerated implementations of BNNs are *only* tailored for desktop platforms. Due to architecture differences, mere porting of such implementations to *mobile devices* yields suboptimal performance or is impossible in some cases. In this paper, we propose PhoneBit, a GPU-accelerated BNN inference engine for Android-based mobile devices that fully exploits the computing power of BNNs on mobile GPUs. PhoneBit provides a set of operator-level optimizations including locality-friendly data layout, bit packing with vectorization and layers integration for efficient binary convolution. We also provide a detailed implementation and parallelization optimization for PhoneBit to optimally utilize the memory bandwidth and computing power of mobile GPUs. We evaluate PhoneBit with AlexNet, YOLOv2 Tiny and VGG16 with their binary version. Our experiment results show that PhoneBit can achieve significant speedup and energy efficiency compared with state-of-the-art frameworks for mobile devices.**

I. Introduction

In the past years, deep neural networks (DNNs) have brought great opportunities and revolutions for many intelligence applications such as automated vehicles and natural language processing. A great success of DNN has been achieved in boosting the performance of the classification accuracy. Being ubiquitous, there is an increasing interest in applying DNNs to mobile environments. DNNs cannot only enhance the performance of mobile applications, but also pave the way toward more intelligent uses of mobile devices [11]. Therefore, as an important impetus towards mobile intelligence, many recent developments in deep learning are tightly connected to tasks meant for mobile devices.

Despite the fact that DNNs are highly useful when deployed on high-end devices, deploying DNNs on mobile devices is still a challenging task because DNNs require too much memory and computing power which significantly exceeds the resource capabilities of current mobile devices and will drain out the battery soon. In addition, DNN continues to get deeper and larger. This trend makes the deployment of DNNs on mobile devices even more challenging. As a consequence, there is an increasing research interest in reducing the computational and memory requirements of DNNs [17].

This work was supported in part by the National Natural Science Foundation of China (NSFC) under Grant 61702085 and Grant 61772123, in part by grant JCYJ20180507182508857.

Binary neural networks (BNNs) proposed in [3] have been demonstrated as a promising solution for mobile computing due to their significantly reduced model size and the complexity of arithmetic operations. Using bit-wise operations to replace floating-point operations, BNN can achieve a significant model compression as well as speedup on accelerating the inference at the cost of small accuracy loss. Despite these results are highly promising, GPU-accelerated BNN inference engine and its highly optimized implementation are still not available yet for mobile devices. While there exists a number of designed deep learning frameworks [7], [18], [1], [12] for mobile devices, most of them are however designed for accelerating general CNN models with full or half precision [5]. In the original paper of BNN [3], [15], *only* a proof-of-concept implementation has been provided to show the performance of BNN. In the implementation, binary weights and activations in BNN are still represented by floating-point values for proof-of-concept purposes. Recently, Pedersoli et al. [13] designed an BNN inference library which was written in CUDA and tailored *only* for desktop platforms. However, because of architecture differences, mere porting of such libraries to *mobile devices* yields suboptimal performance or is even impossible in some cases.

In this paper, we present PhoneBit, a GPU-accelerated BNN inference engine for Android-based mobile devices that explores both software and hardware-level optimization opportunities of BNNs on mobile GPUs. PhoneBit provides a highly optimized framework for execution of BNNs that explores efficient BNN operators to apply high-level optimizations on mobile devices. Using these operator optimizations, an inference of BNN can be effectively executed with minimal memory footprint during run-time. PhoneBit is implemented as a stand-alone inference engine for BNNs with OpenCL, a GPU programming language supported by most mobile GPU architectures. To enable real-time and highly efficient BNN implementations on mobile GPUs, we present the detailed parallelization optimization practices when implementing PhoneBit with OpenCL. The contributions of this work can be summarized as follows:

- We present PhoneBit framework for exploiting computing power of BNNs on mobile GPUs in a systematic way.
- We propose a set of operator-level optimizations including locality-friendly data layout, bit packing with

978-3-9819263-4-7/DATE20/©2020 EDAA

vectorization and layers integration for efficient binary convolution.
- We provide a detailed implementation and parallelization optimization for PhoneBit to optimally utilize the memory bandwidth and computing power of mobile GPUs.

We evaluated PhoneBit using real world workloads on two mobile platforms with different SoCs: Snapdragon 820 and Snapdragon 855. Experimental results show that PhoneBit can achieves up to 38× speedups and 89× energy efficiency over existing GPU-based frameworks.

II. Related Work

Binary Neural Networks. BNNs are deep neural networks that use binary values for activations and weights, instead of full precision values. BNNs are good candidates for deep learning implementations on FPGAs and ASICs due to their bitwise efficiency. Accelerating BNN inference at hardware level has been intensively investigated in [10], [9]. However, these solutions are not versatile as they are hardware-specifically dependent. In general, mobile computing requires the versatility on the applications, i.e., applications developed for one vendor's platform can also execute on other vendors' platforms. Regarding software-based solutions, BMXNet [22] is an open-source BNN library based on MXNet, which is mainly designed for desktop platforms. Recently, Pedersoli et al. [13] presented an BNN inference library Espresso which was written in CUDA. In [13], the optimization for binary matrix multiplication kernels was discussed. However, the more advanced optimizations such as layer integrations are not presented yet. In addition, Espresso is *only* tailored for desktop/server GPUs and cannot be applied on resource constrained *mobile devices* because of architecture differences.
Deep Learning Framework for the Mobile. There are many studies designed deep learning frameworks for mobile devices. These frameworks include TensorFlow Lite (TFLite) from Google [18], Caffe2 from Facebook [1], PaddlePaddle Lite from Baidu [12], CoreML from Apple [2], and CNNdroid [7]. However, most of the existing mobile libraries are limited to CPU/GPU acceleration for the computations of general CNN models with full or half precision [5]. Currently, TFLite supports 8-bit network quantization *only* for CPUs. Network quantization on GPUs is not supported yet in TFLite. To the best of our knowledge, such GPU-accelerated BNN libraries are not available yet on mobile platforms.

III. Background

A. Mobile GPU Architecture

Now, emerging programming models such as OpenCL have been supported by mobile GPUs. GP-GPU computing in mobile devices becomes possible. However, due to the strict area and power constraints, mobile GPUs have some big differences from server/desktop GPUs [7]. In mobile GPUs, the ability of powerful parallel computation relies on parallel computing units called compute units (CUs). Each CU is composed of several parallel ALUs. Fig. 1 illustrates the architecture overview of Qualcomm Snapdragon 855 SoC that integrates a mobile GPU named Adreno 640 consisting of 2 CUs. Each CU in Adreno 640 GPU contains 192 ALUs with SIMD operations.

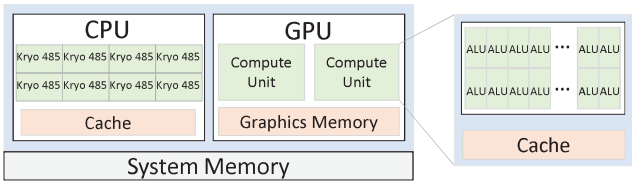

Figure 1. Architecture overview of Qualcomm Snapdragon 855 SoC integrated with Kryo 485 CPU and Adreno 640 GPU.

Compared with server/desktop GPUs, mobile GPUs have limited processing resources. A state-of-the-art desktop GPU usually have thousands of ALUs and thousands of KBytes of L1/shared memory on-chip in total. Meanwhile, a state-of-the-art mobile GPU like Adreno 640 produced by Snapdragon only has 384 ALUs and 1024 KBytes graphics memory on-the chip. Under such limited computational resources, it is more challenging to build high-performance programs on mobile GPUs, especially for DNN-based applications when compared to desktop GPUs.

B. Binary Convolution Operation

In BNN, the weights are binarized, which drastically reduces memory size and accesses. To achieve efficient convolution operators, dot production in binary convolution operation can be replaced by xor (for multiplications) and $popcount$ (for accumulations), as presented in Eqn. (1). $\vec{A}$ and $\vec{B}$ are binary vectors of length Len while a_i and b_i are the i_{th} binary elements in $\vec{A}$ and $\vec{B}$, respectively.

$$\vec{A} \cdot \vec{B} = Len - 2 \times (popcount(xor(a_i, b_i))) \quad (1)$$

The input of the convolution layer typically comes as images, which conflicts with the requirement of binary input for the binary convolution layer. In PhoneBit, we follow [3] and split the input I into bit-planes I_i. Then, binary convolution is operated on bit-plane I_i and binary weights W. The output s can be obtained by summing up the convolution of all bit-planes.

$$s = \sum_{n=1}^{8} 2^{n-1} < I_i \cdot W > \quad (2)$$

where I_i is the bit-plane after splitting and $<>$ denotes a binary convolution operation.

IV. The Overview of PhoneBit Framework

This section provides a high level overview of PhoneBit framework that significantly simplifies the deployment of BNN on mobile devices. In the PhoneBit framework, nearly all common types of BNN layers are well supported such as convolution, pooling, batch normalization, and dense (i.e. fully connected) layers. We implement PhoneBit as a stand-alone GPU-accelerated inference engine for BNNs using OpenCL on mobile devices. Fig. 2 summarizes execution steps involved in deploying trained BNN models on mobile devices. PhoneBit first takes a model trained by existing BNN training frameworks and provides a set of scripts to transform it into the

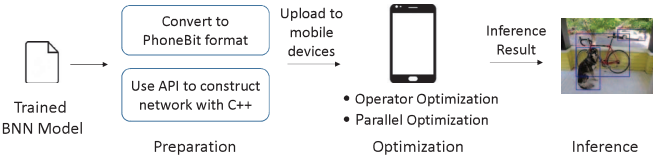

Figure 2. Execution steps in deploying trained BNN models on mobile devices.

compressed PhoneBit format. Once the trained BNN model is generated and uploaded to mobile phones, the BNN model can be deployed on mobile phones in a few simple steps. A code snippet of the corresponding C++ inference interface is presented in Fig. 3. In a few lines of code, a user can call the PhoneBit APIs to construct networks under the PhoneBit framework and get a deployable module for BNN using GPUs on mobile phones.

```
//(1) Initalization (Preparing Input and Packing Weights)
...
//(2)Construct Network and Forwarding
//layer 1
conv1.bforward_S(&img, &padding_size1, &kernel_size1, &
    stride1, &w1, &bn1);
pool1.forward_S(&conv1.out, &size1, &stride_p1, MAX);
//layer 2
conv2.bforward64_S(&pool1.out, &padding_size2, &
    kernel_size2, &stride2, &w2, &bn2);
pool2.forward_S(&conv2.out, &size2, &stride_p2, MAX);
//add more layers
.....
```

Figure 3. Code snippet for the BNN deployment using PhoneBit

V. Optimizing Binary Operations

PhoneBit framework exploits efficient binary neural network operators to apply high-level optimizations on mobile phones. It implements many operator-level optimizations, including: locality-friendly data layout, which enables efficient memory access; bit packing with vectorization, which packs bits on channel direction; and layer integration, which integrates multiple operations in different layers together. In this section, we will present these optimization techniques for efficient binary neural network operators in detail, with a focus on binary convolution.

A. Channel Compression

In BNN, matrix multiplications can be efficiently executed by utilizing the binary instructions xor and $popcount$ when working with binarized weights and input data. In PhoneBit, we pack the bits in channel dimensions into multiple compressed bytes (or other data type such as *short* with 16-bit and *int* with 32-bit), and then perform a compressed convolution on compressed tensor and filters.

1) Data Layout: In BNN, we require to compress the bits in channel dimensions to achieve compressed tensors for efficient convolution operations. This determines that the channel dimension is the best choice along which bit packing should be conducted. Rather than using the NCHW data layout, the default setting adopted in mainstream deep learning frameworks such as Caffe [1] and Torch [20], PhoneBit use NHWC data layout to achieve efficient bit-packing. In PhoneBit, we use $T = R^{H*W*C}$ to represent the tensor, where $h \in [0, H]$, $w \in [0, W]$, and $c \in [0, C]$ represent the tensor dimensions of height, width, and channel, respectively. A minor-to-major dimension order of tensor A is row-major order with interleaved channels. By using internal data layouts for BNN execution, channel-packing can be effectively performed along the channel dimension with efficient memory access when unrolling a tensor, as well as efficient bit operations when performing convolution computation.

2) Bit Packing: Mobile GPU uses a SIMD model with various additional constraints that make it more efficient. OpenCL supports built-in vector data types [19] which are defined with the type name, i.e., $uchar$, $ushort$, $uint$, and $ulong$ followed by a literal value n that defines the number of elements in the vector. Supported vectorization values of n are 2, 4, 8, and 16. To maximize vector unit utilization, PhoneBit uses these built-in OpenCL APIs in its kernel code for efficient bit-wise operations. PhoneBit supports parallel bit-wise operations in different parallelization granularity from 8-bit to 1024-bit[1]. To achieve massive parallelism of BNN executions, PhoneBit selects the optimal bit packing strategy and computing kernel according to channel dimensions.

B. Layer Integration

In general, convolutional operations in BNN consist of three layers, including binary convolution, batch-normalization (BN), binarization layers. The layer overflow that transmits data among different layers requires expensive data movement operations that copy and write tensors for several times during the inference. In addition, deploying a BN layer will further introduce floating-point computations and increase the computation burden. To resolve these issues, we propose a layer integration technique that combines multiple operators in these layers into a single kernel without saving the intermediate results in memory. This optimization can greatly reduce execution time for mobile GPUs.

Now, we will mathematically show how these three layers can be aggregated as one integration operator. For ease of presentation, we denote x_1 as the result of binary convolutional computation without considering the bias b and x_2 as the output of binary convolution layer. Therefore, we have:

$$x_2 = x_1 + b \tag{3}$$

In the BN layer, let x_3 be the output of a BN layer, γ and β denote the trainable parameters, μ and σ are estimated from the sample mean and sample variance of mini-batch, respectively. The transformation can be presented as follows:

$$x_3 = \gamma \cdot \frac{x_2 - \mu}{\sigma} + \beta \tag{4}$$

Substituting x_2 in Eqn. (4) using Eqn. (3). x_3 can be represented as follows:

$$x_3 = \frac{\gamma}{\sigma} \cdot (x_1 - \xi) \tag{5}$$

where ξ is determined as[2]:

$$\xi = \mu - \frac{\beta \cdot \sigma}{\gamma} - b \tag{6}$$

[1]Using ulong16 to achieve 1024-bit vectorization

[2]According to [8], the convolutional channel with γ = 0 can be pruned. Thherefore, we do not consider the case with γ = 0.

Note that ξ can be computed in the off-line stage without increasing the runtime computation burden. Then, x_3 is binarized to the output x_4 after a BN layer. Therefore, we have:

$$x_4 = \begin{cases} 1 & \text{if } x_3 \geq 0 \\ 0 & \text{otherwise} \end{cases} \tag{7}$$

From Eqn. (5), we know the parameters of γ and ξ determine the sign of x_3. Therefore, we can easily transform the transformation functions above as an integrated one, as presented in Eqn. (8).

$$x_4 = \begin{cases} 1 & \text{if } x_1 \geq \xi \text{ and } \gamma > 0 \\ 0 & \text{if } x_1 < \xi \text{ and } \gamma > 0 \\ 1 & \text{if } x_1 \leq \xi \text{ and } \gamma < 0 \\ 0 & \text{if } x_1 > \xi \text{ and } \gamma < 0 \end{cases} \tag{8}$$

VI. PhoneBit Implementation and Parallelization Optimization

In this section, we provide a highly optimized GPU-accelerated implementation for PhoneBit. We implement PhoneBit as a stand-alone inference engine for BNNs with OpenCL, a GPU programming language supported by most mobile GPU architectures. Major optimization steps, such as workload optimization, avoiding branch divergence, memory optimization, are provided to demonstrate the effectiveness of the optimization practices when implementing PhoneBit.

A. Memory Optimization

Optimization techniques for the computational efficiency of the binary operations have been discussed in Section V. PhoneBit has intricate data structures and their memory behavior has significant impact on the performance. In this section, we focus on the memory efficiency of PhoneBit and present the following memory optimization techniques used in PhoneBit.

1) Vectorized Load/Store: We first use vectorized load/store functions in OpenCL to greatly reduce the number of load/store operations in PhoneBit. These build-in functions in OpenCL, supported by most mobile GPUs, allow the hardware to bulk load/store data to/from memory. Such strategy typically takes advantage of the spatial and temporal locality properties of the programs. To achieve better bandwidth utilization, PhoneBit loads/stores the data in chunks of multiple bytes (e.g., 128-bit) using vectorized load/store functions [21].

2) Coalesced Memory Access: To maximize memory access bandwidth, the GPUs try to coalesce memory accesses from work items in the same wavefront into a single memory request if these memory accesses have good spatial localities. To achieve the optimized performance, we apply the adjacent memory-based tile optimization [21] to achieve the coalesced memory access. In addition, for binary convolution operations on the NHWC data layout, the binary operations in each region of the feature map are directly applied to the packed bits that are stored in memory consecutively. Therefore, memory coalesced accesses can also be achieved along the lowest channel dimension in PhoneBit.

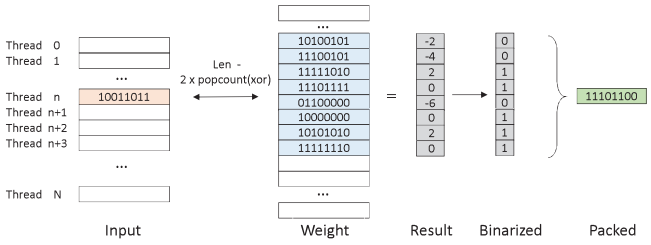

Figure 4. One thread computes 8 filters, binarizes 8 results and pack into one byte. Len is the length of vector.

3) Memory Latency Hiding: Latency hiding refers to the process of overlapping memory operations with computation to maximize the utilization of memory and compute resources. Latency hiding is one of the most powerful characteristics of GPU for efficient parallel processing and enables GPU to achieve high throughput. In PhoneBit, we interleave the memory load and computation into different threads such that memory load and computation can be swapped in a pipelined manner to achieve characteristic of the latency hiding. The pipeline can hide most memory access overheads and almost fully utilize compute resources.

B. Workload Optimization

In BNN, we pack the binary output of convolution kernels to achieve the compressed feature maps. One straightforward approach is to use one thread to conduct the calculation of one convolution kernel and use a byte to store the binarized output. Then, one additional thread is applied to pack the bits along the channel dimensions. However, such implementation requires several individual threads to perform channel compression procedures. The computations of convolution, BN, binarization layers cannot be aggregated for efficient binary operators. Furthermore, this implementation also introduces additional cost for thread synchronization.

To resolve the issues above, PhoneBit uses a single thread to calculate the binary output of 8 convolution kernels and pack them in a compressed byte, as shown in Fig. 4. In this strategy, the data are stored in private memory of the thread and can be rapidly accessed. Therefore, packing operations can be integrated to avoid unnecessary memory access operations as well as thread synchronization cost. However, due to the limitation of private memory size, one thread cannot load too much data once a time. Especially when the channel number is too large, private memory of one thread cannot load the required data for the computation of 8 convolution kernels. In PhoneBit, we assign the computing thread with proper workload according to the number of channels. When the channel number is no greater than 256, PhoneBit will conduct the computations for 8 convolution kernels and integrate packing operations inside. Otherwise, packing operations will be processed separately without integration.

C. Avoiding Branch Divergence

Generally, GPUs are not efficient when work items in the same wave follow different execution paths. For divergent branches, some work items may have to be masked, resulting

Table I
MOBILE DEVICES

Device	SOC	Memory	OS	OpenCL Version	ALUs in GPU
Xiaomi 5	Snapdragon 820	3GB	Android 7.0	2.0	256
Xiaomi 9	Snapdragon 855	8GB	Android 9.0	2.0	384

in lower GPU occupancy [21]. In PhoneBit, the computations of BNN layers have been transformed as Eqn. (8), which requires to use divergent checks to determine the output. In PhoneBit, we present novel software-based optimizations to convert divergent check operations to fast logic operations. We first use the truth table to represent the determination of Eqn. (8). Based on this representation, the following logic function Eqn. (9) can be computed by the laws of boolean algebra and be further simplified by Karnaugh maps [6].

$$x_4 = (A\ xor\ B)\ or\ C \tag{9}$$

where A, B, and C denote the boolean variables of $x_1 < \xi$, $\gamma > 0$, $x_1 = \xi$, respectively, which can be rapidly determined using OpenCL build-in functions $isless$, $isgreater$ and $isequal$. By using fast logic operations in Eqn. (9), divergent checks in PhoneBit can be avoided.

VII. EXPERIMENT

Experiment Setup. PhoneBit is evaluated on two mobile devices with different SoCs: Snapdragon 820 and Snapdragon 855. The detailed hardware configurations are shown in Tab. I. We evaluate PhoneBit with three classic neural network models, namely, AlexNet network for CIFAR10 dataset, YOLOv2 Tiny network for VOC2007 dataset and VGG16 network for CIFAR10 dataset. We conduct the experiments to evaluate the efficiency of PhoneBit by the extensive comparison with state-of-the-art frameworks:

- **CNNdroid [7]:** a RenderScript-based framework [16] for paralleling computations of full-precision CNN across CPUs and GPUs. Both CPU- and GPU-based executions are evaluated for performance comparison. However, as indicated in [4], even executing in GPU-based execution, RenderScript is not always using GPUs on all the devices – sometimes it is still running on a CPU only.
- **TensorFlow Lite (TFLite) [18]:** a lightweighted deep learning framework developed by Google for mobile devices. TFLite supports accelerators on both CPUs and GPUs. Currently, TFLite only support lower to 8-bit quantization specification on CPUs and does not support model quantization on GPUs [18]. Due to these limitations, three implementations, including one GPU-based and two CPU-based executions with and without quantization, are evaluated for performance comparison.

We implement forwarding of the benchmark networks on the mobile devices. The metrics of accuracy, model size, runtime, power and energy consumption are evaluated and compared to demonstrate the efficiency of PhoneBit.

Accuracy and Model Size. First, we run three binarized benchmark networks using PhoneBit, and compare their accuracy and model size with full-precision networks. The compared results are shown in Tab. II. The model size of networks

Table II
MODEL SIZE(MB) AND PRECISIONS

	Model Size(MB)		Precision(%)	
	Full-precision CNN	BNN	Full-precision CNN	BNN
AlexNet	249.5	16.3	89.0	87.2
YOLOv2 Tiny	63.4	2.4	57.1	51.7
VGG16	553.4	32.1	92.5	87.8

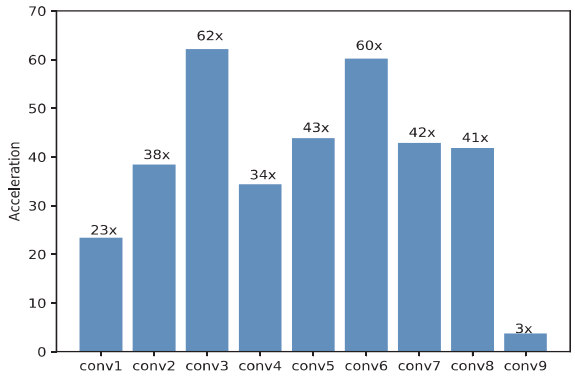

Figure 5. Performance improvement brought by PhoneBit BNN implementations, with counterpart float-value operators in CNNdroid (GPU based execution) as the baseline(1x), tested on Snapdragon 855 platform.

is critical for space-constrained mobile devices. For all three benchmark networks, PhoneBit can achieve significant compression rate with an acceptable accuracy loss.The model size of PhoneBit is on average 19.6× smaller than full-precision networks. For large network model such as VGG16, PhoneBit can achive 17.2× storage efficiency with 4.7% accuracy loss when compared to full-precision networks.

Runtime Performance. Next, we will demonstrate the efficiency of PhoneBit by comparing runtime performance with state-of-the-art frameworks. Tab. III shows the runtime performance of three benchmark networks for the compared frameworks under different hardware settings. From Tab. III, we can make the following observations: (1) PhoneBit brings significant speedups over all of the compared frameworks on both mobile devices. Compared with CNNdroid [7], PhoneBit on average gains 794× and 35× speedups on CPU based and GPU based executions, respectively. On TFLite, PhoneBit can on average bring 12×, 15×, and 6× faster speedup over CPU based, GPU based and CPU quantization based executions, respectively. (2) Stability issues have been observed on CNNdroid and TFLite frameworks. For large-scale network VGG16, CNNdroid runs out of memory (OOM) on both CPU and GPU implementations while TFLite runs into crash (CRASH) and fails to produce the results on GPU based executions. In contrast, PhoneBit can work flawlessly with all benchmark networks with faster inference speed.

We also investigate how our parallelism optimization methods can explore the massive parallelism on various convolutional layers of networks. To examine the impact of PhoneBit on various layers of networks, we measure run-time of each layer in YOLOv2-Tiny where the first layer comes as 8-bit integers and the last layer is a full precision layer for final float type output. In PhoneBit, multiple operators in binary convolutional, BN, binarization layers are combined into a

Table III
AVERAGE RUNTIME(MS) IN SNAPDRAGON 820 AND SNAPDRAGON 855 PLATFORMS USING DIFFERENT FRAMEWORKS

	Snapdragon 820						Snapdragon 855					
	CNNdroid		Tensorflow Lite			PhoneBit	CNNdroid		Tensorflow Lite			PhoneBit
	CPU	GPU	CPU	GPU	CPU Quant		CPU	GPU	CPU	GPU	CPU Quant	
AlexNet	8243	766	143	CRASH	103	22.9	5621	369	87	CRASH	24	9.8
YOLOv2 Tiny	51313	1483	669	468	503	42.1	23144	845	306	430	88	22.6
VGG16	OOM	OOM	2607	CRASH	1907	152.3	OOM	OOM	932	CRASH	252	73.8

Table IV
ENERGY CONSUMPTION PER IMAGE FRAME FOR YOLOV2 TINY NETWORK ON SNAPDRAGON 820 PLATFORM

	CNNdroid		Tensorflow Lite			PhoneBit
	CPU	GPU	CPU	GPU	CPU Quant	
Watts(mW)	914	573	626	540	452	225.67
Efficiency(FPS/W)	0.02	1.18	2.39	3.97	4.40	105.26

single kernel for efficient convolutional computation, using layer integration technique presented in Section V-B. Fig. 5 shows the performance improvement over CNNdroid with GPU based execution[3] on the integrated layers for YOLOv2-Tiny network. For the middle binary layers from conv2 to conv8, PhoneBit can bring $45\times$ (up to $62\times$) acceleration over CNNdroid on average. Such acceleration benefits from not only operator optimization techniques presented in Section V but also parallelization optimization techniques presented in Section VI. On conv1, PhoneBit only gains about $23\times$ speedups because of additional process of splitting input integer into bit-planes as we described in Section III-B. On the last layer conv9 with full precision, PhoneBit still gains about $3\times$ acceleration over CNNdroid due to the fact of using SIMD operation on build-in dot product function *dot* in OpenCL.

Energy Efficiency. Energy efficiency is a major design concern for energy-restricted mobile devices. Now, we demonstrate the energy efficiency of PhoneBit by reporting power consumption and energy efficiency under different frameworks. The Trepn Power Profiler [14] is an on-target power and performance profiling tool for Android mobile devices. We use Trepn Power Profiler to measure the power consumption on Snapdragon 820 platform[4]. Tab. IV shows the power consumption and energy efficiency for YOLOv2 Tiny network. From Tab. IV, we can see that PhoneBit can achieve significant power efficiency when executing YOLOv2 Tiny network. PhoneBit consumes around $226mW$ power and obtains 105.26 FPS per watt. Compared with CNNdroid and TFLite implementations, PhoneBit brings $4\times$–$1218\times$ faster execution speed, $2\times$– $4\times$ lower power dissipation and $24\times$–$5263\times$ better performance per power efficiency.

VIII. CONCLUSION

In this paper, we propose a GPU-accelerated BNN inference engine PhoneBit for Android-based mobile devices. In PhoneBit, we propose a set of operator-level optimization techniques to fully explore computing power of BNNs on mobile GPUs. To enable real-time and highly efficient BNN implementations on mobile GPUs, we present the detailed parallelization optimization practices when implementing PhoneBit with OpenCL. On the evaluations of popular neural networks, PhoneBit achieves significant speedups and energy efficiency when compared with state-of-the-art frameworks for mobile devices.

[3]TFLite provides a static compiled library in Android and therefore cannot measure run time in single layer

[4]Trepn Profiler on Snapdragon 855 platform cannot obtain the profiling data of battery power.

REFERENCES

[1] Caffe2. https://caffe2.ai/. Accessed Aug. 12, 2019.

[2] CoreML. https://developer.apple.com/machine-learning/. Accessed Aug. 12, 2019.

[3] M. Courbariaux et al. Binarized neural networks: Training deep neural networks with weights and activations constrained to+ 1 or-1. *CoRR*, 2016.

[4] A. Ignatov et al. AI benchmark: Running deep neural networks on android smartphones. *CoRR*, 2018.

[5] Z. Ji. Hg-caffe: Mobile and embedded neural network gpu (opencl) inference engine with fp16 supporting. *CoRR*, 2019.

[6] M. Karnaugh. The map method for synthesis of combinational logic circuits. *Transactions of the American Institute of Electrical Engineers, Part I: Communication and Electronics*, 1953.

[7] S. S. Latifi Oskouei et al. Cnndroid: Gpu-accelerated execution of trained deep convolutional neural networks on android. In *Proceedings of the 2016 ACM International Conference on Multimedia*, 2016.

[8] Z. Liu et al. Learning efficient convolutional networks through network slimming. In *International Conference on Computer Vision (ICCV)*, 2017.

[9] H. Nakahara et al. A lightweight yolov2: A binarized cnn with a parallel support vector regression for an fpga. In *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)*, 2018.

[10] E. Nurvitadhi et al. Accelerating binarized neural networks: Comparison of fpga, cpu, gpu, and asic. In *2016 International Conference on Field-Programmable Technology (FPT)*, 2016.

[11] K. Ota et al. Deep learning for mobile multimedia: A survey. *Acm Transactions on Multimedia Computing Communications & Applications*, 2017.

[12] PaddlePaddle. https://github.com/PaddlePaddle/. Accessed Aug. 12, 2019.

[13] F. Pedersoli et al. Espresso: Efficient forward propagation for binary deep neural networks. In *The International Conference on Learning Representations (ICLR)*, 2018.

[14] TrepnProfile. https://developer.qualcomm.com/software/. Accessed Aug. 12, 2019.

[15] M. Rastegari et al. Xnor-net: Imagenet classification using binary convolutional neural networks. In *European Conference on Computer Vision (ECCV)*, 2016.

[16] RenderScript. https://developer.android.com/reference/android/renderscript/RenderScript. Accessed Aug. 12, 2019.

[17] V. Sze et al. Efficient processing of deep neural networks: A tutorial and survey. *Proceedings of the IEEE*, 2017.

[18] Tensorflow Lite. https://www.tensorflow.org/lite/. Accessed Aug. 12, 2019.

[19] The OpenCL Specification. https://www.khronos.org/registry/OpenCL/specs/opencl-2.0.pdf. Accessed Aug. 12, 2019.

[20] Torch7. https://github.com/torch/torch7. Accessed Aug. 12, 2019.

[21] H. Wang et al. Opencl optimization and best practices for qualcomm adreno gpus. In *Proceedings of the International Workshop on OpenCL*, 2018.

[22] H. Yang et al. Bmxnet: An open-source binary neural network implementation based on mxnet. In *Proceedings of the 25th ACM International Conference on Multimedia*, 2017.