Profiling and Optimizing Deep Learning Inference on Mobile GPUs

Shiqi Jiang, Lihao Ran, Ting Cao, Yusen Xu, and Yunxin Liu
Microsoft Research
{shijiang,v-lihran,ticao,v-yusxu,yunliu}@microsoft.com

ABSTRACT

Mobile GPU, as the ubiquitous computing hardware on almost every smartphone, is being exploited for the deep learning inference. In this paper, we present our measurements on the inference performance with mobile GPUs. Our observations suggest that mobile GPUs are underutilized. We study the inefficient issue in depth and find that one of root causes is the improper partition of compute workload. To solve this, we propose a heuristics-based workload partitioning approach, considering both performance and overheads on mobile devices. Evaluation results show that our approach reduces the inference latency by up to 32.8% on various devices and neural networks.

KEYWORDS

Deep Learning Inference, Mobile GPU, Workload Partition

ACM Reference Format:
Shiqi Jiang, Lihao Ran, Ting Cao, Yusen Xu, and Yunxin Liu. 2020. Profiling and Optimizing Deep Learning Inference on Mobile GPUs. In *ACM SIGOPS Asia-Pacific Workshop on Systems (APSys '20), August 24–25, 2020, Tsukuba, Japan.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3409963.3410493

1 INTRODUCTION

With the rapid growth of AI techniques, intelligent edge computing is attracting more and more attentions. Mobile GPUs, for the general availability on mobile devices and more compute power compared to mobile CPUs, are being exploited for on-device deep learning (DL) inference.

However, the performance of DL inference on mobile GPUs is not well studied yet. This paper presents a measurement study which shows the mobile GPU is underutilized. We therefore conduct preliminary research to analyze and solve the issue to improve inference performance on mobile GPUs.

We first study the hardware utilization of mobile GPUs in depth. The profiling results show that during the inference process, although the GPU shader cores are mostly busy, the arithmetic processing units inside cores are still underutilized. Furthermore, we find that the workload partition plays a critical role in the final performance. However, current partition methods used in DL inference frameworks, such as exhaustive search or look-up table, all have obvious deficiencies.

Some works have been done for workload partitioning on server GPUs [8, 9]. Unfortunately, they are inappropriate for mobile GPUs due to the different hardware features [16, 19] such as resource constraints. To this end, we propose a heuristics-based workload partitioning approach for DL inference on mobile GPUs. Based on the hardware and DL-framework characteristics, we extract several heuristic rules, which can filter out partitions with poor performance and keep well-performed partitions.

We implement and integrate the proposed approach with TensorFlow Lite [6]. The early-stage evaluation shows 11% inference speedup on average. Particularly, for models designed for mobile devices e.g. MobileNet [11], the improvement is up to 32.8%. Besides, the proposed approach is very lightweight, which is implemented in less than 100 lines of code. It may be migrated to other inference engines easily.

2 DL INFERENCE ON MOBILE GPUS

We conduct measurements to study the performance of DL inference on mobile GPUs, taking the CPU inference performance as the comparison.

2.1 Study Setup

We use TensorFlow Lite GPU Delegation (TFLite GPU) [6, 15] as the inference engine. We choose popular mobile System-on-Chips (SoC) equipped with dominant mobile GPUs, Arm Mali and Qualcomm Adreno for our measurements. They have more than 79% of market share in 2019 [1], and thus we believe they could represent the performance of mainstream mobile GPUs. In Table 1, we detail the devices used in our

APSys '20, August 24–25, 2020, Tsukuba, Japan
© 2020 Association for Computing Machinery.
ACM ISBN 978-1-4503-8069-0/20/08...$15.00
https://doi.org/10.1145/3409963.3410493

Specification	Hikey 970	Vivo X30	Google Pixel 3XL	Xiaomi Mi 9
SoC	Kirin 970	Exynos 980	Snapdragon 845	Snapdragon 855
CPU	Cortex A73+A53	Cortex A77+A57	Kryo 385	Kryo 485
GPU	Mali-G72	Mali-G76	Adreno 630	Adreno 640
GPU Core Unit	12	5	2	2
GPU Cache Size (KB)	512	256	128	128
Memory Size (MB)	2,048	2,048	1,775	3,752
Memory Bandwidth (GBPS)	18.9	13.2	22.6	25.23
GPU Peak GFLOPS	220.2	174.1	429.1	530.4
CPU Peak GFLOPS	134.4	121.6	146.2	214.8

Table 1: Experimental platforms.

measurements and their specifications. We also profile their peak compute power based on our real tests.

We consider popular deep neural networks which are widely deployed on mobile devices. We select 6 networks and their variants, which are MobileNetsV1 [11], VGG-16/19 [17], ResNet-101/152 [10], InceptionV3 [18], NASNet [20] and DenseNet-121/201 [12]. The networks are converted into TFLite format. We do not apply the quantization and 32-bit floating point (FP32) arithmetic is used.

We make use of TFLite Benchmark tool[1] to profile the performance. The benchmark tool takes a TFLite model and initializes the runtime for the CPU or mobile GPU. Then it automatically generates random inputs and executes repeatedly for 100 runs according to our settings. Finally the aggregated statistics are provided. On each device, we first run selected networks on the CPU using all of its cores, then on the mobile GPU. In order to avoid the overheating problem, we make the device sleep for 30 seconds to cool down between every subsequent runs.

2.2 Results

Fig. 1 illustrates the averaged inference latency while running different deep neural networks using mobile GPU and CPU on each device. Overall, the mobile GPU inference shows a 1.9× speedup on average, compared to the CPU inference.

On Adreno GPUs, the performance improvement is quite notable, which demonstrates up to 3× speedup, as shown in Fig. 1(a) and Fig. 1(b). However, for small networks, i.e. MobileNetsV1 and NASNet, the performance speedup degrades to only 1.5-1.8×.

On Mali GPUs, as illustrated in Fig. 1(c) and Fig. 1(d), the benefits of using the mobile GPU are fairly limited. For large networks like VGG-16/19, the mobile GPU inference even requires higher latency than the inference with CPU.

Summarized in Table 1, the compute power of mobile GPUs generally is about 2-3× more than CPU. However, our measurements show that, for most of cases, the performance of mobile GPU inference cannot achieve the expected performance gain. The big gap suggests that the full power of the mobile GPU is not unleashed. There might exist inefficient issues with the usage of mobile GPUs.

In addition, the overhead of using mobile GPUs cannot be neglected. Table 2 summaries the initialization time of TFLite GPU on each device while running networks above. In the initialization phase, the inference engine would also involve kernel compilations and find proper configurations for the mobile GPU. Shown in Table 2, the initialization surprisingly spends extremely long time, especially for Mali GPUs, even up to 20 minutes.

Model	Hikey 970	Vivo X30	Google Pixel 3XL	Xiaomi Mi 9
VGG19	1,467,829	262,894	10,092	3,693
VGG16	246,950	209,082	10,099	3,944
Resnet-152	90,820	47,266	3,062	1,641
Resnet-101	26,814	32,964	2,107	1,302
MobileNetsV1	11,421	2,708	656	465
InceptionV3	14,637	19,756	1,613	1,039
DenseNet-201	95,764	22,372	1,735	969
DenseNet-121	73,854	17,921	1,081	716
NASNet	9,609	5,685	2,646	1,858

Table 2: Initialization time (ms) on the mobile GPU of each device while running various deep neural networks.

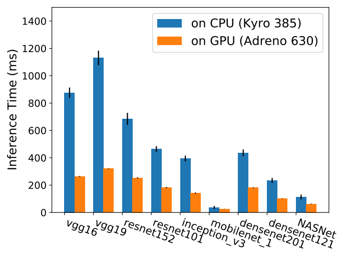
3 UNDERSTANDING THE UNDER-UTILIZATION OF MOBILE GPU

To tackle the inefficient issue of mobile GPU, we perform in-depth measurements and try to identify the root cause. Before getting to the study details, we briefly introduce some background on mobile GPU.

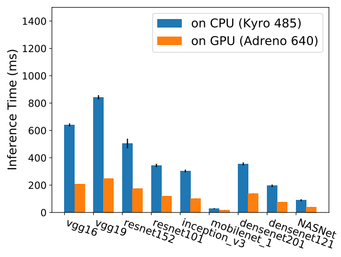
3.1 Background

Fig. 2 illustrates a high-level view of the mobile GPU architecture. There are only one or a few identical shader

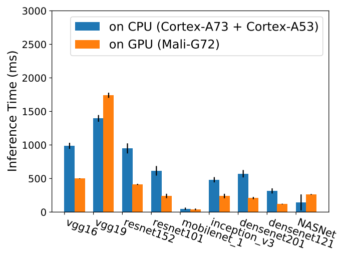
[1]https://git.io/JvySS



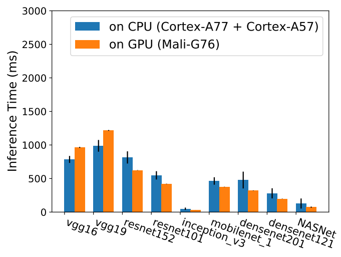
(a) Google Pixel 3XL



(b) Xiaomi Mi 9



(c) Hikey 970



(d) Vivo X30

Figure 1: Inference latency using mobile CPU and GPU on each device for various deep neural networks.

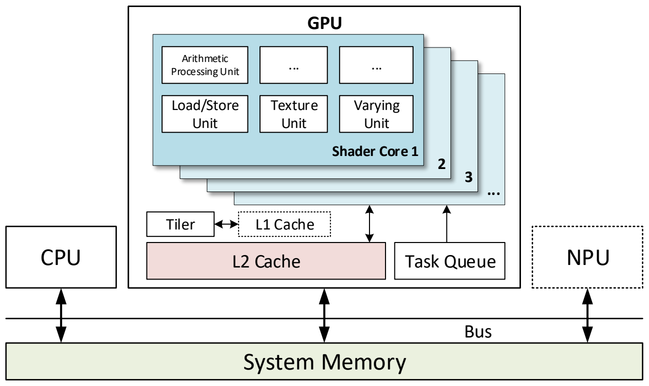


Figure 2: High-level view of mobile GPU architecture.

cores, which are designed to execute both graphics rendering and compute workloads. A shader core consists of fixed function units such as texture and varying units, as well as programmable ones, i.e. arithmetic processing units (APU), which are used for DL inference.

A mobile GPU normally maintains a pair of task queues, one for graphics rendering tasks and the other for compute tasks. A scheduler fetches tasks from each queue, breaks into chunks with proper sizes and distributes across shader cores. The scheduling strategy is usually implemented differently. For instance, First-In-First-Out (FIFO) policy is applied in some Adreno GPUs [13].

A DL inference engine, like TFLite GPU, often wraps each operator of neural networks as a compute task via programming libraries such as OpenGL ES [2], OpenCL [7] or Renderscript [3]. The task then is executed by multiple parallel threads on the mobile GPU. Every thread runs the same routine, but on different regions of the workload, which is called a work item. Several work items can be further organized into a work group. The shape and size of the work group is called work group size. Each work group is assigned to a particular shader core for execution by the scheduler.

Work items of a work group are executed by function units in parallel. Inside one core, work items can easily share data and states. But inter work group communications are not direct, which bring higher latency. On the other hand, the function units inside a core are limited. Therefore, it is essential to organize proper work items into one core to reduce messaging overheads.

In addition, due to strict chip area and power restrictions, mobile GPUs are much more resources constrained compared to server GPUs. For example, NVIDIA Tesla V100 GPU has 6 M L2 cache, 900 GB/s global memory bandwidth as well as large register files and shared memory on chip [14]. By comparison, as shown in Table 1, Adreno 630 on a Google Pixel 3XL only has 128 KB L2 cache and 22.6 GB/s memory bandwidth. Besides, as illustrated in Fig. 2, the memory on mobile devices is shared among GPU, CPU or NPU. Thus, proper workload partition is particularly important for mobile GPUs to reduce memory accesses and improve performance.

3.2 Identifying Root Cause

To deepen understandings on the inefficient issue discussed in §2, we measure the mobile GPU performance metrics in-depth. The metrics we observe are the utilization of shader core, memory bandwidth and APUs. We make use of Streamline [5] and Snapdragon Profiler [4] to collect these metrics on Hikey 970 and Google Pixel 3XL for Mali and Adreno GPUs, respectively.

First, we measure the utilization of shade core, by the proportion of average shade core active cycles in total GPU active cycles. As shown in Fig. 3, when models are running, shader cores are often busy. The averaged utilization is 91.2% on Adreno and 89.8% on Mali GPUs. It suggests that compute workload have been well distributed across all share cores.

Then, we measure the memory bandwidth utilization. Illustrated in Fig. 3, the bandwidth is mostly surplus. For large networks such as VGG-16/19, the utilization is about 21% on Mali and 49% on Adreno. The utilization gets even lower for small networks, i.e. MobileNetsV1. On tested devices our observation suggests that the memory bandwidth limitation

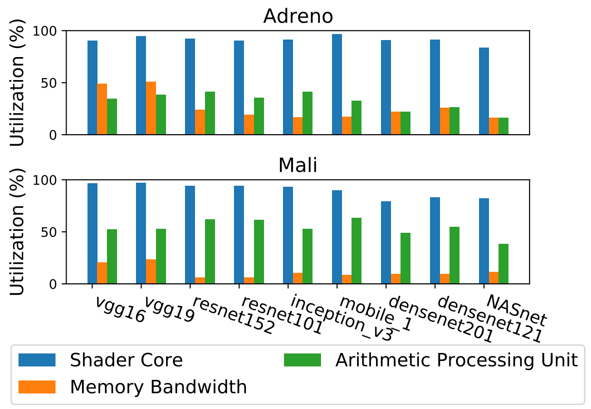


Figure 3: In-depth performance metrics while running various deep neural networks on Hikey 970 (Mali) and Google Pixel 3XL (Adreno).

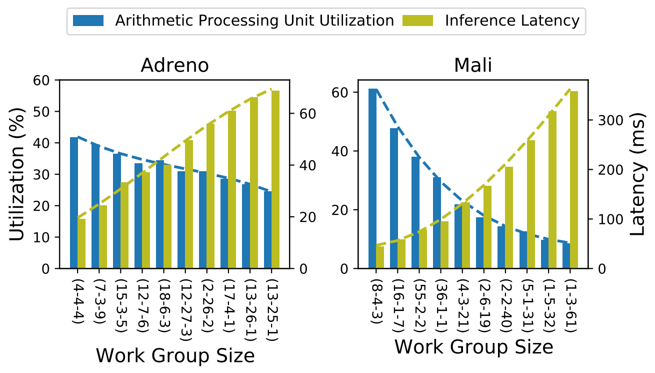


Figure 4: APU utilization and inference latency on Hikey 970 (Mali) and Google Pixel 3XL (Adreno) while running Conv with different work group sizes.

is not the root cause of the inefficient issue for the selected neural networks.

We next measure the APUs inside shader cores. We count the proportion of average active cycles of APUs in total active cycles of a shader core as the APU utilization. Shown in Fig. 3, we notice although shader cores are often busy, APUs inside cores still are underutilized. On Mali the average utilization is 54.3%. On Adreno the average APU utilization is around 32%, which is even lower.

Work items are executed by APUs inside a shader core in parallel. Ideally shader core should spend as much time as possible on computing, which leads to the high APU utilization. But some of APUs might be bounded by cache misses. Then the external memory access would keep APUs waiting, which leads to the low APU utilization. To avoid this issue, we could organize work items that access neighboring data together with the proper size and assign to one shader core. Meanwhile there are several cores on the mobile GPU. Therefore, in order to keep the high utilization inside cores and across cores, a proper work group size should be selected.

Thus, we further study how work group sizes impact the utilization of APUs and the inference latency. To facilitate understanding, we break down the neural network into operators and execute the single operator on both the Adreno and Mali GPU. We pick the typical operator convolution (Conv), since Conv is considered as one of the most time-consuming operators. We select 3×3 as the kernel size, and set 56×56×256 as the input shape, which is a common Conv shape shared across various networks [10, 17]. We enumerate all possible work group sizes and then execute Conv with them. The inference latency and the APU utilization are collected.

For each device we select 10 work group sizes and their corresponding results to showcase. Illustrated in Fig. 4, we observe that different work group sizes have significant impacts on the APU utilization on both Adreno and Mali GPUs. The inference latency is also closely correlated with the utilization. The higher utilization leads to the lower latency. More specifically, on the Mali GPU, compared with a non-optimal work group size (1-3-61), a finest one (8-4-3) can bring up to 7.1 times improvement in the utilization of APUs. Meanwhile the inference speedup on Mali is about 8.02 times. On the Adreno GPU, the improvement in the utilization is about 1.8×, which makes the inference gets 3.58× faster.

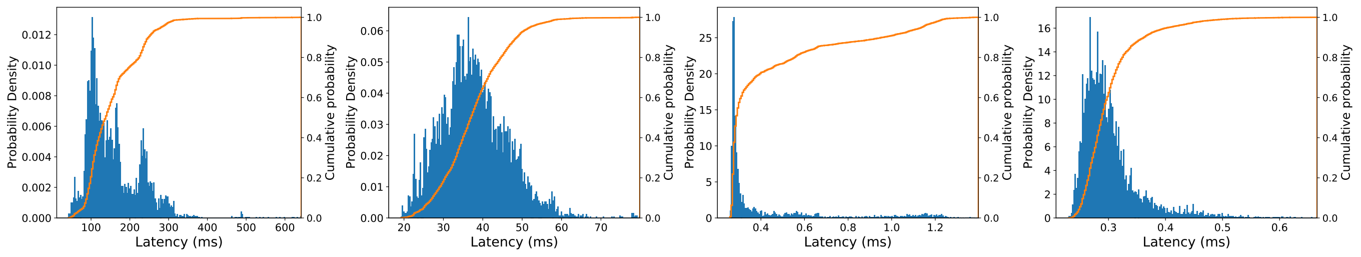
We also conduct the experiments above on other devices listed in Table 1. We obtain similar results. To summarize our study, we find when running various deep neural networks, although mobile GPU shader cores are often busy, APUs inside cores are still underutilized. It in turn leads to the inefficient issue. To cope with it, a proper work group size should be selected carefully.

4 WORK GROUP SIZE SELECTION ON MOBILE GPUS

Selecting the optimal work group size on GPUs is not new. Some works have been done on server GPUs but are inappropriate for mobile GPUs, which are either too heavy for mobile devices or not efficient enough (detailed in §5). To this end, we propose a heuristics-based workload partition approach, considering both inference performance and overheads. Based on heuristics from hardware and DL characteristics, we propose several rules that can guide us to find out the optimal work group sizes fully utilizing the mobile GPU.

4.1 Heuristics-based Workload Partition

First, we introduce our observations on hardware and DL characteristics. Fig. 5 shows the latency distribution of all work group sizes from the Conv and depth-wise convolution



(a) Conv on Mali. **(b) Conv on Adreno.** **(c) Dw-Conv on Mali.** **(d) Dw-Conv on Adreno.**

Figure 5: The latency distribution of Conv and Dw-Conv using all possible work group sizes on Hikey 970 (Mali) and Google Pixel 3XL (Adreno); input shapes are 56×56×256 for Conv and 56×56×128 for Dw-Conv; kernel size is 3×3.

(Dw-Conv) on the Mali GPU and Adreno GPU. The shapes are selected from VGG-16/19 and Inception V3 [17, 18], etc. Overall, the latency follows the Gaussian and mixed Gaussian distribution, but the distributions are quite different across operators and devices. The observation suggests the specific rules should be designed accordingly.

From the distributions, we also observe there are still many work group sizes having the similar performance with the best. For instance, if a candidate's latency is closed to the best within 15%, we call it is a competitive candidate. We find that 69.4% of work group sizes are competitive on Mali, while around 30% on Adreno for Dw-Conv. For Conv, there are also about 9% competitive candidates. Based on this observation, instead of finding the best work group size, we could select from competitive candidates at a low cost. Next, we discuss the heuristic rules to guide to select.

Given an operator of the neural network, we notate its output shape as (x', y', z'). The inference engine might apply the vectorization scheme. Notating the vectorization factors on each dimension as (v_x, v_y, v_z), then the problem space can be defined as $(x, y, z) = (\lceil x'/v_x \rceil, \lceil y'/v_y \rceil, \lceil z'/v_z \rceil)$. A work group size is notated as (w_x, w_y, w_z). For each (w_x, w_y, w_z), we check if it follows all of the proposed heuristic rules.

Rule I, fully utilizing shader cores. We should keep the high utilization of shader cores. To use all cores on the target device, the problem space should be partitioned according to the maximum number of shader cores:

$$\lceil x/w_x \rceil \cdot \lceil y/w_y \rceil \cdot \lceil z/w_z \rceil \equiv 0 \pmod{N_{sc}}, \quad (1)$$

where N_{sc} is the number of shader cores of the target mobile GPU. We can obtain N_{sc} via public APIs [2, 7].

Rule II, fully utilizing APUs. We should leverage all APU computation capacities inside a core. Each work group should have enough work items that can be distributed across all APUs. Thus, we make the size of each group should be a multiple of the number of APUs:

$$w_x \cdot w_y \cdot w_z \equiv 0 \pmod{K_{ac}}, \quad (2)$$

where K_{ac} is the maximum number of work items that can be simultaneously executed on a shader core. In practice, K_{ac} depends on not only the number of APUs, but the operator kernel's register footprint and register file size as well. We can also obtain K_{ac} via public APIs [2, 7].

Rule III, fully utilizing cache lines. We should use the L2 cache in the efficient way. For each data access, we make data block just fill the whole cache line:

$$w_i \cdot d_i \cdot U \equiv 0 \pmod{W_{cache}}, i \in \{x, y, z\} \quad (3)$$

where d_i is the data block which is read each cycle along the tiling dimension i. In practical, d_i is related to the vectorization scheme and the particular data layout used in the inference engine. U is the size of floating point arithmetic. W_{cache} represents the width of cache line in bytes. In our experiment, $i = x$ and we extract d_x from TFLite GPU where d_x is 8. U is 4 since FP32 is used. W_{cache} could be obtained via the public API [7].

Rule IV, minimal problem space padding. Most of mobile GPUs only support the unified size for all work groups. Thus, the problem space might be padded if it is not divisible by the partition. The padded workload obviously bring extra costs. To avoid such padding we have the rule:

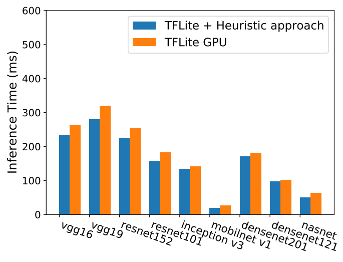
$$x \equiv 0 \pmod{w_x}, \ \ y \equiv 0 \pmod{w_y}, \ \ z \equiv 0 \pmod{w_z}, \quad (4)$$

Rule V, minimal data accesses. Given a compute task, we should reuse the loaded data as much as possible by minimizing the total data accesses, which leads to the high utilization of mobile GPUs. We notate the data access number of a work item as D_k. D_k should be a function related to the particular kernel k and w_x, w_y, w_z, x, y, z:

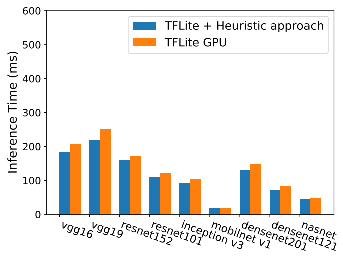
$$D_k = g(w_x, w_y, w_z, x, y, z) \quad (5)$$

The total data access number TD_k should be $TD_k = w_x \cdot w_y \cdot w_z \cdot D_k$. By adjusting w_x, w_y and w_z to minimize TD_k, we could get the work group size rule.

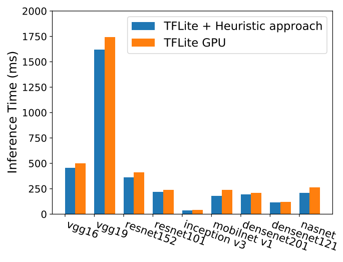
To obtain D_k, we analyze the source code of the kernel k, and manually count the number of data accesses. Currently



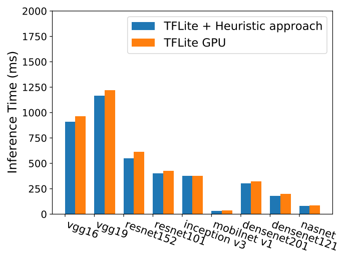
(a) Google Pixel 3XL.



(b) Xiaomi Mi 9.



(c) Hikey 970.



(d) Vivo X30.

Figure 6: Inference latency using TFLite GPU with and without our approach on each device while running various deep neural networks.

the kernel CONV and Dw-Conv are analyzed. We have the following rule:

$$\text{For Conv: } w_x \cdot w_y \approx 2 \cdot w_z, \tag{6}$$

$$\text{For Dw-Conv: } w_x \cdot w_y \approx min(C_x \cdot C_y,\ C_t), \tag{7}$$

where C_x, C_y are the maximum number of work items that can be specified in the dimension x and y of a work group. C_t is the maximum total number of work items in a work group. C_x, C_y and C_t are related to the kernel k and hardware constraints, which can be obtained by public APIs [7].

The heuristics we discuss consider the characteristics of the hardware (Rule I-III), the inference engine (Rule III, IV) as well as the particular kernel implementations (Rule V). We have more than these five rules, but we find these are most efficient. We evaluate the rules by the importance score, which is defined as

$$S_r = \frac{1}{M} \sum_{n=1}^{M} \frac{f(W_p) - f(W_n)}{f(W_p) - f(W_b)}, \tag{8}$$

where M is the number of work group sizes that follow the rule r. Given an operator, $f(w)$ is the inference latency using the work group size w. W_n, W_p, W_b stand for the evaluated work group size, the work group size having the highest latency and the lowest latency, respectively. Therefore, for an operator, the higher S_r is, the more good work group sizes the rule r can filter out.

We apply Equation 8 on the dataset used in Fig. 5. Table 3 summarizes the obtained importance scores. In run time, we iteratively apply these rules until the work group size is selected. There may be multiple candidates left. In that case, we would randomly select one, because we believe they have closed performance.

4.2 Preliminary Evaluation Results

We implement the proposed approach and integrate it into latest TFLite GPU 2.1 on Android devices. Our approach is very lightweight. It contains less than 100 lines of code and

Operator	Rule I	Rule II	Rule III	Rule IV	Rule V
Conv	0.74	0.81	0.80	0.78	0.82
Dw-Conv	0.88	0.90	0.86	0.90	0.99

Table 3: The importance scores of Rule I-V for Conv and Dw-Conv on tested devices.

it is easy to be migrated onto other inference engines. Currently Conv and Dw-Conv are supported. For other operators the predefined work group size is used.

We evaluate our approach in terms of the inference latency and the APU utilization. We use the neural networks mentioned in §2.1 and the devices listed in Table 1. We use the performance of TFLite GPU as the baseline.

Fig. 6 illustrates the inference latency using TFLite GPU with and without our approach on each device. Overall, our approach could reduce 11.2% inference latency on average compared to TFLite GPU. Small networks i.e. MobileNetsV1 are widely deployed in mobile applications. We could achieve up to 32.8% inference-latency reduction. For large networks i.e. VGG-16/19, up to 14.5% improvement could be obtained.

On the Adreno GPU, our approach gets notable results. Up to 32.8% performance improvement is achieved and the averaged improvement is 13%. On the Mali GPU, the averaged improvement is about 8%. TFLite uses the exhaustive search for Mali GPUs. The results also suggest that the exhaustive search cannot guarantee to find the optimal work group size, there might exist further issues.

We evaluate the APU utilization improvement as well. Compared to the default TFLite GPU, we obtain 12% utilization improvements on average. On the Mali GPU, the utilization improves up to 18% and 8% on Adreno.

Our approach is very lightweight. The initialization involving our approach costs around 2.7 seconds on average. For small networks, the initialization can be finished within 1 second, which is acceptable for real mobile applications.

5 RELATED WORK

A number of works have been done to tune the work group size towards improving GPU performance, which mainly fall into the following three categories.

Exhaustive search. This method enumerates all possible work group sizes and tests on the target device in run time. The one with the lowest latency is selected. However, iterative tests require significant time especially on resource limited devices. In addition the long exhaustive search also easily triggers the overheating problems, leading to the performance degradation. TFLite applies this method for Mali GPUs. Shown in Table 2, up to 20 minutes are spent where nearly three thousand work group size candidates are tested.

Look-up table. This method obtains optimal settings for popular hardware offline and saves into the look-up table. In run time the selected work group size is read from the table. However, current mobile GPUs usually can be configured by manufactures [16], and thus there exist lots of hardware variants. It is hard to cover all of them. Besides that, the optimal work group size depends not only hardware, but also operators' kernel implementations. TFLite uses this method for Adreno GPUs. Shown in Fig 3, the APUs are still underutilized.

Prediction model. Machine learning based methods are proposed too [8, 9]. By selecting features and training the model, the best work group size and corresponding performance are predicted. But due to the diversity and variety of mobile devices and neural networks, it is too hard to collect necessary training data and build a generic model.

6 CONCLUSION AND FUTURE WORK

In this paper, we take a first look at the inefficient issue of deep learning inference with mobile GPUs. From measurements, we find that although shader cores of the mobile GPU are busy, arithmetic processing units inside cores are still underutilized. The proper work group size is critical for high mobile GPU utilization. To this end, we propose a heuristic-based approach, considering both inference performance and overheads. The early-stage results show our approach could speed up the inference by up to 32.8%.

The rules we propose are based on heuristics from the characteristics of hardware, inference engines and kernel implementations. Currently we manually analyzed Conv and Dw-Conv. In the future, we would leverage the code analysis techniques to automatically extract rules from the kernel source code, supporting more kernels. During our study, we also notice other inefficient system issues with mobile GPU inference. One of them is the overheating problem. How to achieve the efficient and stable inference within the thermal constraint may be an interesting problem to work on.

REFERENCES

[1] 2019. Q2 2019 Smartphone and Tablet GPU Market Share. https://www.businesswire.com/news/home/20191118005549/en/.

[2] 2019. The Standard for Embedded Accelerated 3D Graphics. https://www.khronos.org/opengles/.

[3] 2020. RenderScript Overview. https://developer.android.com/.

[4] 2020. Snapdragon Profiler. https://developer.qualcomm.com/software/snapdragon-profiler.

[5] 2020. Streamline Performance Analyzer. https://developer.arm.com/tools-and-software/embedded/arm-development-studio.

[6] 2020. TensorFlow Lite, ML for Mobile and Edge Devices. https://www.tensorflow.org/lite.

[7] 2020. The Open Standard for Parallel Programming of Heterogeneous Systems. https://www.khronos.org/opengcl/.

[8] Chris Cummins, Pavlos Petoumenos, Michel Steuwer, and Hugh Leather. 2016. Autotuning OpenCL Workgroup Size for Stencil Patterns. In The 6th International Workshop on Adaptive Self-tuning Computing Systems (ADAPT).

[9] T. L. Falch and A. C. Elster. 2015. Machine Learning Based Auto-Tuning for Enhanced OpenCL Performance Portability. In 2015 IEEE International Parallel and Distributed Processing Symposium Workshop.

[10] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. In Proceedings of the IEEE conference on computer vision and pattern recognition. 770–778.

[11] Andrew G Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. 2017. Mobilenets: Efficient convolutional neural networks for mobile vision applications. arXiv preprint arXiv:1704.04861 (2017).

[12] Gao Huang, Zhuang Liu, Laurens van der Maaten, and Kilian Q. Weinberger. 2017. Densely Connected Convolutional Networks. 2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR) (2017).

[13] Gang Huang, Mengwei Xu, Felix Xiaozhu Lin, Yunxin Liu, Yun Ma, Saumay Pushp, and Xuanzhe Liu. 2017. ShuffleDog: Characterizing and Adapting User-Perceived Latency of Android Apps. IEEE Transactions on Mobile Computing 16, 10 (2017), 2913–2926.

[14] Zhe Jia, Marco Maggioni, Benjamin Staiger, and Daniele Paolo Scarpazza. 2018. Dissecting the NVIDIA Volta GPU Architecture via Microbenchmarking. CoRR abs/1804.06826 (2018).

[15] Juhyun Lee, Nikolay Chirkov, Ekaterina Ignasheva, Yury Pisarchyk, Mogan Shieh, Fabio Riccardi, Raman Sarokin, Andrei Kulik, and Matthias Grundmann. 2019. On-Device Neural Net Inference with Mobile GPUs. (2019). arXiv:1907.01989 http://arxiv.org/abs/1907.01989

[16] Arm Limited. 2019. Arm Mali GPU OpenCL Developer Guide. Technical Report.

[17] Karen Simonyan and Andrew Zisserman. 2014. Very deep convolutional networks for large-scale image recognition. arXiv preprint arXiv:1409.1556 (2014).

[18] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jon Shlens, and Zbigniew Wojna. 2016. Rethinking the Inception Architecture for Computer Vision. 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR) (2016).

[19] Qualcomm Technologies. 2017. Qualcomm Snapdragon Mobile Platform OpenCL General Programming and Optimization. Technical Report.

[20] Barret Zoph, Vijay Vasudevan, Jonathon Shlens, and Quoc V. Le. 2018. Learning Transferable Architectures for Scalable Image Recognition. 2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition (2018).