On-Device Augmented Reality with Mobile GPUs

Juhyun Lee, Nikolay Chirkov, Ekaterina Ignasheva, Yury Pisarchyk, Mogan Shieh,
Fabio Riccardi, Raman Sarokin, Andrei Kulik, and Matthias Grundmann
Google Research
1600 Amphitheatre Pkwy, Mountain View, CA 94043, USA
{impjdi,chirkov,eignasheva,ypisarchyk,moganshieh,fricc,sorokin,akulik,grundman}@google.com

Abstract

Reliable computer vision is a prerequisite for augmented reality (AR) applications; in particular when applied to mobile devices. Many state-of-the-art vision techniques employ deep neural networks. However, inference is a compute-intensive task and solely using mobile CPU can be difficult due to limited computing power, thermal constraints, and energy consumption. App developers and researchers have begun exploiting hardware accelerators to overcome these challenges. Recently, device manufacturers are adding neural processing units into high-end phones for on-device inference, but these account for only a small fraction of hand-held devices. In this paper, we present how we leverage the mobile GPU, a ubiquitous hardware accelerator on virtually every phone, to achieve real-time AR effects for both Android and iOS devices.

1. Introduction

Augmented reality (AR) heavily relies on the computer's ability to understand the environment, so that meaningful overlays and annotations can be autonomously applied. Recent computer vision research largely built on the success of deep convolutional neural networks which in turn are employed by many AR applications. AR apps on hand-held devices desire to run aforementioned neural net inference *on the device*, primarily for latency reasons.

On-device inference, however, is a non-trivial task. Despite recent advances in mobile hardware technology and efforts to efficiently run deep networks on mobile devices [3, 11, 4, 6, 7], mobile CPUs continue to be less powerful than those found in servers. Running deep net inference on a mobile device means adding a significant compute-intensive task to the CPU. Fully utilizing mobile CPUs comes with unwanted costs, *e.g.* increased energy consumption leading to shorter battery life and thermal throttling resulting in slower computation.

Our primary goal is a fast inference engine for typical AR applications with wide coverage of supported devices. By leveraging the mobile GPU, a ubiquitous hardware accelerator on virtually every phone, we can achieve real-time performance for various AR effects that employ deep nets. Major hardware manufacturers are publishing software development kits for inference on their devices [1, 5, 8, 10]. Also, popular machine learning frameworks have limited mobile support, *e.g.* Caffe2 [2] and MACE [12] only work on vendor-specific GPU architectures. TensorFlow Lite (TFLite) leverages the mobile GPU with OpenGL ES 3.1+ for Android devices and Metal for iOS 9+ devices.

This paper presents the techniques we adopt for TFLite GPU and how we achieve an average acceleration of 2–9× for various deep networks on GPU compared to CPU inference. We first describe the general mobile GPU architecture and GPU programming, and then present how TFLite GPU allows real-time on-device AR effects.

2. Mobile GPU Inference

Initialization In contrast to CPU inference, GPU inference engines require initialization involving shader compilation and optimization by the driver. The cost of this process depends on network size and may take from few milliseconds to seconds, but is incurred once and not again for subsequent runs until the cache memory is invalidated for various reasons (application update,, device reboot, *etc.*)

Data Layout Most modern GPUs use a homogeneous coordinate [9] system which represents points in space with coordinates (x, y, z, w). For any $w \neq 0$, homogeneous coordinates (x, y, z, w) represent a point $(x/w, y/w, z/w, 1)$ in a 3D space. This allows transformations to be represented in the form of 4D matrix multiplications. In this way, GPUs are ideally suited for efficient computation and memory load/store of 4-element vectors.

In TFLite GPU, a $[H, W, C]$ tensor is split into 4-channel slices which are then stored sequentially in memory. If the number of channels is not divisible by 4, it is padded with zeroes. This memory layout, called PHWC4 (Figure 1), op-

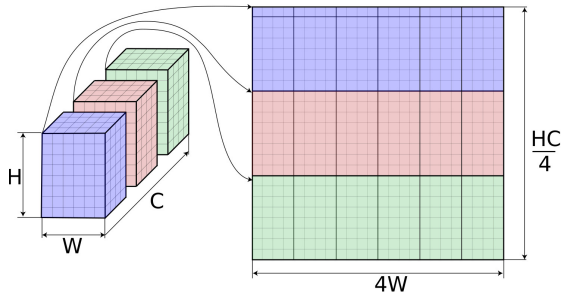

Figure 1. Example of PHWC4 memory layout (best viewed in color). A tensor of shape $(H{=}8, W{=}6, C{=}12)$ is split into 4-element slices of size $(H, W, 4)$ which are stored sequentially as a continuous 2D array of size $(HC/4{=}24, 4W{=}24)$.

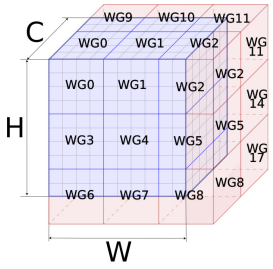

Figure 2. Compute shader execution grid $(X{=}12, Y{=}12, Z{=}8)$ built upon the tensor shape $(H{=}10, W{=}10, C{=}6)$ shown in blue (best viewed in color). Work group size $(x{=}4, y{=}4, z{=}4)$ highlighted as cubes with bold lines. Each cell represents a FP32 value.

timally reduces cache misses in the graphics architecture. This is tightly coupled with how compute threads are executed on the GPU, which defines the order of computation, and, more importantly, the order of memory reads.

Shader Program Optimization In the GPU inference engine, operators exist in the form of shader programs. The shader programs eventually get compiled and inserted into the command queue and the GPU executes programs from this queue without synchronization with the CPU.

To reduce the number of shader programs in the command queue, we consolidate them into meaningful aggregates while maximizing parallelism and well-defined data dependencies. We apply optimization techniques such as operator fusion, inlining parameters and/or objects, and specialization by kernel size/hardware type/driver version.

The source code for each program is generated and then compiled. This compilation step can take a while, from several milliseconds to seconds. Typically, app developers can hide this latency with displaying a splash screen while loading the model. Once all shader programs are compiled, the engine is ready for inference.

GPU Inference The input tensors are reshaped to the PHWC4 format, if their tensor does not have exactly 4 channels. Shader programs for each operator are linked by binding resources such the its input/output tensors, weights, *etc*. and dispatched, *i.e*. inserted into the command queue. The GPU driver then takes care of scheduling and executing all shader programs in the queue, and makes the result available to the CPU via synchronization. For maximum performance, it is best to avoid this synchronization, and preferably, stay on the GPU context if real-time processing is needed. The most ideal scenario is a camera provides an RGBA texture directly to TFLite GPU with the resulting output of the network being directly rendered to the screen.

Work Groups: GPU Threading Units A compute task consists of a shader program and a grid pair. A shader program is a piece of code re-used by every thread, and a grid is an abstraction of a 3D mesh of physically executed threads. The global grid is made up of repeated work groups of constant shape (x, y, z) and has a total dimension (X, Y, Z), that is a multiple of these work groups.

When calculating the compute grid size of an operation, we use the shape of one of the output tensors as a basis. The grid may be larger than the actual output tensor, because we expand it to sizes in multiples of 4 due to GPUs working efficiently for those sizes. This may cause the creation of threads without real workload, but this is faster than working with misaligned grid sizes. This is depicted in Figure 2. Blue grids show threads with real workload, and red grids show stub threads without real workload.

Optimizations are focused on neighboring threads *within* a work group. Our PHWC4 layout allows neighboring threads to hit the same cache line when requesting data for input tensors. Our investigation revealed the following execution order for threads inside a work group: For each work group channel, each row is sequentially picked in order from the first to last, starting across W, then H, and finally C. Ordering of work group execution is likewise sequential and follows the same pattern.

Work Group Size Selection The work group size for executing shader programs defines the group of threads which share data inside the work group. Picking the right work group size can result in increased performance, whereas the opposite can lead to unexpected slowdowns. Unfortunately, tuning the work group size is difficult because GPU internals, *e.g*. thread execution group ("wave") size, are not available to the user. Selecting the optimal work group size thus becomes an exhaustive search.

We focused on optimizing the work group size for CONV_2D and DEPTHWISE_CONV, as these make up roughly 90% of the workload for convolutional networks. We use a gradient descent approach to converge on a stable optimum work group size. Work groups from the Table 1 are currently used in TFLite GPU.

Figure 3. Real-time AR effects using TFLite GPU (best viewed in color). Upper row: (a) Left: Fast Face Detection; 200Hz on Pixel 1. (b) Center: 2D Contour; 50Hz on Pixel 2. (c) Right: 3D Face Mesh; 280Hz on iPhone 7. Lower row: (d) Left: Normal Map Prediction; 10Hz on Pixel 2. (e) Center: Background Segmentation; 40Hz on Pixel 2. (f) Right: Multi Segmentation; 40Hz on Pixel 2.

Adreno GPU Model	CONV_2D	DEPTHWISE_CONV
630	$(4, 8, 4)$	$(4, 4, 8)$
540	$(8, 2, 2)$	$(8, 8, 2)$
510	$(8, 4, 4)$	$(8, 4, 4)$

Table 1. Optimal work group sizes for some Adreno GPUs.

3. On-Device AR with Mobile GPU

TFLite GPU is used by a set of major AR applications and AR developer APIs on mobile phones. Figure 3 lists a variety of AR use cases running in or near real-time on various phones. Figure 3 (d) demonstrates the preprocessing of surface finding for an AR app at 10Hz which would not have been possible on the CPU, because inference is too slow (3.5Hz) for real-time AR. TFLite GPU supports arbitrary neutral network architectures as long as the shader code exists for each op in the network. TFLite GPU is a natural fit for many AR applications, since they involve registering and overlaying virtual objects on a live camera feed, all of which can be performed on the GPU without incurring the penalties of memory copies to the CPU.

4. Discussion

We presented the architectural design of our mobile GPU neural network inference engine TFLite GPU. In this section, we outline a couple of architectural considerations that can make neural networks more GPU-friendly.

First, RESHAPEs are significantly more expensive on the GPU than on the CPU. The network itself will learn the weights regardless of the RESHAPE op, thus it is best to skip the operator entirely if a RESHAPE operation was inserted just for convenience of the architect.

Second, if the camera of the mobile device can produce RGBA data rather than RGB, it should be now apparent that using the former can avoid a conversion, *i.e.* memory copy, from RGBA to RGB. Similarly, if the mobile device can render a 4-channel tensor, *i.e.* RGBA, directly, that can be a better choice than the RGB counterpart. This choice benefits not just the graph input/output, but also its intermediate tensors. Finally, since we know that a tensor of shape $[B, H, W, 5]$, for instance, is twice as expensive as $[B, H, W, 4]$, but about the same as $[B, H, W, 8]$, then the architect can tune around those 4-channel boundaries rather than trying to optimize on other boundaries.

References

[1] Arm Ltd. Compute Library. https://developer.arm.com/ip-products/processors/machine-learning/compute-library. [Online; accessed 8-April-2019]. 1

[2] Facebook Inc. Caffe2. https://caffe2.ai. [Online; accessed 8-April-2019]. 1

[3] Andrew G Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications. *arXiv preprint arXiv:1704.04861*, 2017. 1

[4] Jonathan Huang, Vivek Rathod, Chen Sun, Menglong Zhu, Anoop Korattikara, Alireza Fathi, Ian Fischer, Zbigniew Wojna, Yang Song, Sergio Guadarrama, et al. Speed/Accuracy Trade-offs for Modern Convolutional Object Detectors. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 7310–7311, 2017. 1

[5] Huawei Technologies Co., Ltd. HiAI Engine. https://developer.huawei.com/consumer/en/devservice/doc/2020315. [Online; accessed 8-April-2019]. 1

[6] Andrey Ignatov, Nikolay Kobyshev, Radu Timofte, Kenneth Vanhoey, and Luc Van Gool. DSLR-Quality Photos on Mobile Devices with Deep Convolutional Networks. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 3277–3285, 2017. 1

[7] Andrey Ignatov, Radu Timofte, Thang Van Vu, Tung Minh Luu, Trung X Pham, Cao Van Nguyen, Yongwoo Kim, Jae-Seok Choi, Munchurl Kim, Jie Huang, et al. PIRM Challenge on Perceptual Image Enhancement on Smartphones: Report. In *European Conference on Computer Vision*, pages 315–333. Springer, 2018. 1

[8] MediaTek Inc. What is MediaTek NeuroPilot? https://www.mediatek.com/blog/what-is-mediatek-neuropilot. [Online; accessed 8-April-2019]. 1

[9] August F. Möbius. *Der baryzentrische Calcül*. 1827. 1

[10] Qualcomm Inc. Snapdragon Neural Processing Engine SDK. https://developer.qualcomm.com/docs/snpe. [Online; accessed 8-April-2019]. 1

[11] Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. MobileNetV2: Inverted Residuals and Linear Bottlenecks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 4510–4520, 2018. 1

[12] Xiaomi. MACE. https://github.com/XiaoMi/mace. [Online; accessed 8-April-2019]. 1