Optimizing DNN Operators on Mobile GPUs

Brian Park
bcpark@ncsu.edu
North Carolina State University
Raleigh, North Carolina, USA

Abstract

Performance and tuning of Deep Neural Networks (DNN) operators are often important for fast and efficient performance. The importance becomes more critical when you target a mobile device, which is constrained on compute, memory, and power resources. Thus, performance and optimization must be carefully considered. Here, we add support for two DNN models for image classification using Xiaomi's Mobile AI Compute Engine (MACE) on Android devices. MACE is an open source project where DNNs are compiled to run on Android devices, with backends for CPU, GPU, and NPU supported. Specifically, we implement a fast grouped convolution for GPU to complete support for RegNet and an optimized channel shuffle for GPU to complete support for ShuffleNet V2+. The implementation and framework is open-sourced, and can be downloaded at https://github.com/briancpark/csc766-project

Keywords: DNNs, compiler, mobile, HPC, GPU, computer architecture

ACM Reference Format:
Brian Park. 2023. Optimizing DNN Operators on Mobile GPUs. In *Proceedings of (CSC 766 Course Project).* ACM, New York, NY, USA, 7 pages. https://doi.org/XXXXXXX.XXXXXXX

1 Introduction

Although MACE is classified as an engine and interpreter, it can also be seen as a compiler framework to compile and run DNNs on Android device. This is most similar to other frameworks from their respective vendors such as Apple's CoreML, Meta's PyTorch Mobile, Google's TFLite, Tencent's NCNN, Alibaba's MNN, and many more. One of the benefits of MACE is that it is open-sourced, so its implementation can be learnt from and improved upon by the open source community. Because the platform is targeted towards Android smartphones, with first class support of XiaoMi devices, the microkernels are implemented NEON intrinsics and OpenCL to enable SIMD and SIMT parallelism on CPU and GPU respectively. This report will focus on optimization of GPU kernels via OpenCL.

CSC 766 Course Project, May 2023, Raleigh, NC

ACM ISBN XXXXXXXXX…$0.00
https://doi.org/XXXXXXX.XXXXXXX

2 Motivation, Objectives, and Related Work

Here, we give a quick overview of the operators to be implemented and what needed to be done to give end-to-end support of the DNN models to be compiled. The two models are specifically ShuffleNet V2+ Small and RegNet (200M). These are the smallest versions of the models that should be ideal to run on a mobile smartphone. These are both models that are trained and optimized over accuracy on the ImageNet dataset, a 1000 label image dataset popularized for image classification benchmarks [4].

2.1 ShuffleNet V2+ Small

ShuffleNet V1 is a model architecture that has been highly popularized over its competitive accuracy and efficiency over ResNet and MobileNet [6]. Later, improvements for practical guidelines for efficient design and high performing convolutional neural networks have been proposed in ShuffleNet V2 [3]. As observed from the ShuffleNet designers, convolution operations take the most amount of FLOPs as well as compute time in a CNN. An issue with traditional ConvNets are that they are often deep and have bottlenecks if they want to be designed for mobile devices. Thus, the goal of version 2 of ShuffleNet was to introduce a *channel shuffle* operation that can enable information communication between different groups of channels. This not only improves accuracy, but also improves efficiency as well. Figure 1 visually shows how the channel shuffle operation is performed and how it can improve efficiency in network design as shown from the original ShuffleNet paper [6].

Some variants of ShuffleNet use group convolutions or depth-wise convolutions. MACE does not support the former, group convolutions. Fortunately, the version of ShuffleNet we used does not utilize any group convolutions. Later in RegNet, we'll explain how group convolutions work, so it is possible to compile and run a variant of ShuffleNet with group convolutions. It's also important to note that MACE does support ShuffleNet V2 for CPU and GPU. But its GPU implementation of channel shuffle is only limited to a group size of 4. Thus, we had to implement channel shuffle with a group size of 2 in OpenCL to fully complete support. This was the most straightforward task, and it gave us an introduction

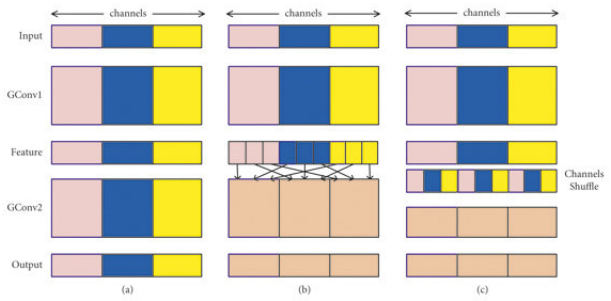

Figure 1. Channel Shuffle Mechanism; (a) two stacked convolutional layers with the same number of groups; (b) input and output channels are fully related when GConv2 takes data from different groups after GConv1; (c) an equivalent implementation to (b) using channel shuffle instead [6]

to how the MACE framework worked when implementing this operation.

2.2 RegNet (200M)

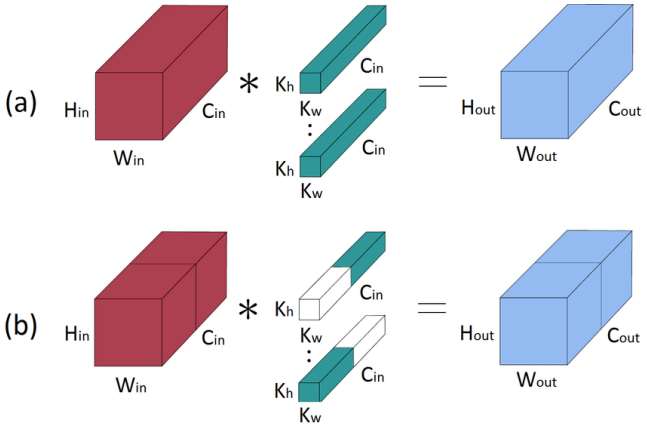

Figure 2. Group Convolution Mechanism; (a) regular convolution (or group convolution with 1 group; (b) Group convolution with 2 groups [1]

RegNet is a work also popularized over its competitive accuracy and efficiency over state-of-the-art models like ResNet, VGG, and AlexNet [5]. Its work focuses on the design principles for efficient and high performing convolutional neural networks, much like ShuffleNet V2. It combines advantages of manual design and Neural Architecture Search (NAS). Thus, its focus is evaluation of network architecture of other competitive networks and ablation studies to figure out strength and flaws in convolutional network design. However, group convolution was not originally implemented in RegNet. It dates way back to AlexNet, when GPUs were very limited by memory and compute resources at the time [1]. The AlexNet author's experimental setup included two GTX 580 GPUs with 3GB of GPU memory. At the time, there was no such concept of NVLink or GPU-GPU communication, thus there was a large overhead of communication between the two GPU devices via CPU-GPU and GPU-CPU communication through the PCI lanes. As a result, the authors were bottle-necked by how much memory and compute a single GPU could perform and proposed a *group convolution* algorithm that is embarrassingly parallel and can be distributed across multiple devices. In this case, they split a regular convolutional layer into multiple groups and split it across multiple GPUs. The convolutions that are done in groups require no communication and are only aggregated in the end when passed to the next layer, making the algorithm for group convolution *communication optimal*. This was written in 2012, and since then GPU hardware has been greatly improved due to the demand of deep learning, with GPU memory reaching a peak of 80GB of high bandwidth memory (HBM) and compute power reaching orders of magnitude in tera-FLOPS. As an effect, group convolutions have become obsolete until its efficiency under compute and memory constraints was required again for mobile CPUs and GPUs, which is where it has been revived in RegNet architecture. Figure 2 shows a visual representation of how the group convolution operation works. Note that depth-wise convolution is when groups are equal to the number of input channels. This was already implemented in MACE, so we carefully studied already existing implementations of regular convolution and depth-wise convolution to implement an optimized group convolution kernel.

3 Implementation

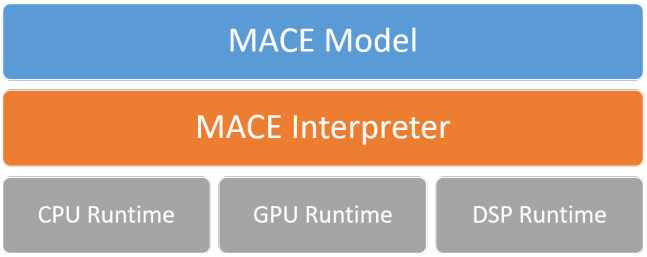

Figure 3. MACE System Architecture

To complete support of ShuffleNet and RegNet, several steps were required in order to complete support in the MACE framework. Here, we give a high level overview of how the operators were implemented under the MACE framework. First was to compile the model to ONNX format, which is a high level representation of a DNN in the semantics of tensors and operators. This can be easily done with PyTorch or TensorFlow models. We chose to compile ONNX files from PyTorch models. Then, we need to convert from ONNX format to MACE ops format, which is another intermediate representation (IR) that the MACE framework

can understand and apply optimizations over. Those optimizations include operator fusion, high level mathematical properties, and allocating which hardware target to run efficiently on. For example, a convolutional layer can be fused with activation layer, to reduce the memory traversal and footprint. If some ops are incompatible on GPU, the MACE framework can temporarily fallback to CPU to at least support the functionality, despite the performance tradeoff. In order to do these optimizations, they were implemented in higher level code base written in Python to generate the MACE IR. The MACE IR is then passed to C/C++ bindings of optimized operators in programming frameworks or libraries such as ARM NEON intrinsics and OpenCL. All of these are eventually lowered to `libmace.so` to integrate to an Android application. In order to evaluate real world performance, understanding how to develop an Android application is needed. But for the focus of this report, we utilize the workflow from MACE to give us some preliminary benchmarks as shown in Figure 4. Particularly, the benchmarking framework gives us detailed op-by-op performance and throughput for us to debug performance and correctness of specific layers and operators.

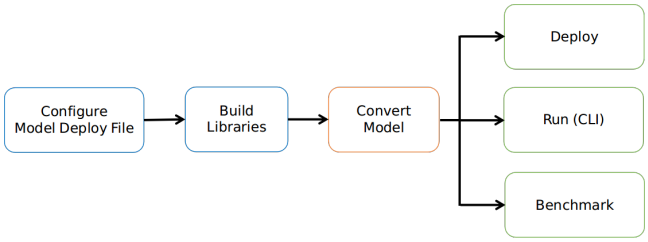

Figure 4. MACE Framework Workflow

4 Experimental Results

We target the XiaoMi 11 Lite. This phone has an Qualcomm SM7150 Snapdragon 732G octa-core CPU, with (2) 2.3 GHz performance cores and (6) 1.8 GHz efficiency cores. This follows a Big Little architecture for the core cluster. It's important to note when evaluated against CPU configurations of the model, only the performance cores are utilized. This is a safety feature, as oversubscribing all the cores could lead to intractability of other applications and tasks that are corunning on the smartphone. Also, the imbalance of the cores' frequency causes load imbalance and is hard to optimally parallelize and synchronize. The GPU is equipped with Adreno 618 GPU, but little is publicly known about the microarchitectural features in order to fully optimize performance. We also only implement and optimize under the assumption under ARM v8 ISA. The ARM v7 ISA is supported in MACE. The main difference between v7 and v8 is that v7 is 32 bit ISA and v8 is 64 bit ISA. But because of the time of doing this project, v8 is a widely adopted ISA. Thus, our implementation is not backward compatible to older devices. It's also important to note that multithreading is implemented by MACE themselves, where they tile and parallelize loops based on their own heuristics from one dimension to up to three dimensions (dimension in terms of nested for loops). Their threading library is wrapped around OpenMP, thus there should be minimal overhead of initializing, creating, and destroying threads.

To keep benchmark consistent, we measure the inference time by an input tensor of size $1 \times 224 \times 224 \times 3$, which is the ImageNet dataset parameters. The output is 1×1000. Note that when comparing against the CPU and GPU configuration, *some operators in GPU configuration may fall back to CPU*. This will be explicitly noted in the results shown, but this is mainly due either because (1) the operator is not fully implemented on the GPU; (2) it is more optimal to run on CPU than on GPU. MACE does not make this explicitly clear when a model is converted under the engine, so we have to guess which choice it makes based on other context. Also MACE has its own heuristic to determine whether to run on CPU or GPU, and it will prefer to stay resident on one hardware target as long as possible. This is because of the overhead of switching between targets and remapping the page table for CPU or GPU. Although the smartphone typically has unified memory, this also affects cache coherency between the CPU and GPU (under the assumption that L1 or L2 cache is not shared between the two). In the end, it's a simple heurstic implemented by MACE, but can also miss some oppurtunities as we will see in the results.

We choose an evaluation over batch size of 1, because in production, it's physically impossible to batch multiple images to image processing and inference pipeline in real-time, unless the developer wants to incur some overhead in predictions to serve to the user. Intuitively, only one image is processed at a time, thus we also optimize the group convolution and channel shuffle operations under the assumption that a batch size 1 will be called frequently. We do not consider the optimization under a batched inference, but it should be supported, although maybe not as performant. Since MACE is meant only for inference, we felt that it was not important to focus on batched inputs, although it would be helpful in other applications other than image classification.

MACE also strictly requires GPU kernels to be implemented in NHWC format, while CPU can be inter-operable between NHWC or NCHW, but prefers NCHW. This is because the OpenCL programming system prefers the NHWC format to take advantage of its image and buffer memory programming model. Because we compiled from ONNX, which prefers a NCHW format, there will be additional `BufferTransform` operators inserted for the GPU congiguration to transpose the ops from NCHW to NHWC for the inputs and outputs.

For the results shown, they are all run under 1000 trials and the average is shown.

4.1 ShuffleNet V2+ Small

ShuffleNet V2+ Small has 16 ops in total that are channel shuffle operations, as shown in Table 1. The channel shuffle operation is already implemented for the CPU and GPU, which GPU only supporting group size of 4. We simply modified the GPU kernel for channel shuffle to add support for group size of 2, which is what all of ShuffleNet V2+ Small has.

For the CPU implementation, it should be noted that it's a scalar implementation. There is no multi-threading involved and it solely uses `memcopy`. Under the hood, it's almost certain `memcopy` is compiled and optimized to use NEON vector instructions. But this operation is purely memory bounded operation with no computations. It's very possible that GPU may not give any significant speedups, or even slower speedups. Typically, GPUs are clocked at lower frequency, with a tradeoff of higher ALU count. Since there is no actual computation happening and the memory is shared between CPU and GPU, it's possible to not achieve speedups. Thus, we observed *slowdowns* for channel shuffle on GPU compared against CPU.

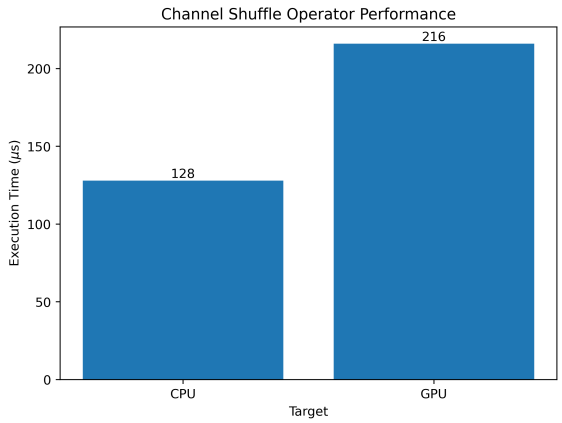

Figure 5. Channel Shuffle Op Performance

4.1.1 Channel Shuffle Performance.

Figure 5 shows the performance of channel shuffle over the whole execution time of ShuffleNet V2+ Small. Unfortunately, there's a slowdown of 0.6× compared against CPU with a runtime of 216 microseconds. The execution time is very negligible in terms of the whole runtime (about 1% of ShuffleNet execution time on CPU). Any improvements does not really bring significant speedups to the whole model. Considering that the channel shuffle operators for both CPU and GPU happen at the magnitude of microseconds, there was not much motivation to improve further for the GPU operator.

4.1.2 End-to-End Performance.

When compiled under GPU configuration, slice and concat ops do not support the operators on the channel axis dimension. Thus, both ops fall back to CPU and run on NCHW format, causing lots of transpose ops inserted to convert back and forth between the two formats. In addition, the last MatMul (GEMM) and Reduce Mean operator fall back to CPU. The reduce operator also falls back due to unsupported axis on the GPU, where reduce mean is only supported height and width axis. This causes an end-to-end slowdown of 0.2× compared against the CPU as shown in Figure 6. Figure 7 shows the breakdown op-by-op. We observe significant improvements in computationally heavy operations such as Conv2d and DepthwiseConv2d. But we see that there is even an additioanl slowdown for operators that fellback on CPU in GPU configuration, which is Concat, Slice, Reduce, and MatMul. We are not sure of the root cause, but we suspect that some formats are not fully optimized on CPU for NHWC format or there is additional overhead between switching hardware targets.

Also note that the breakdown doesn't necessarily add up to the GPU runtime of 60ms, so there is some overhead involved when benchmarking the MACE framework op-by-op.

Table 1. ShuffleNet V2+ Small Channel Shuffle Parameters

Tensor Name	Input/Output Shape
`Transpose_34`	[1,48,28,28]
`Transpose_82`	[1,48,28,28]
`Transpose_130`	[1,48,28,28]
`Transpose_178`	[1,48,28,28]
`Transpose_215`	[1,96,14,14]
`Transpose_263`	[1,96,14,14]
`Transpose_311`	[1,96,14,14]
`Transpose_359`	[1,96,14,14]
`Transpose_407`	[1,96,14,14]
`Transpose_455`	[1,96,14,14]
`Transpose_503`	[1,96,14,14]
`Transpose_551`	[1,96,14,14]
`Transpose_588`	[1,192,7,7]
`Transpose_636`	[1,192,7,7]
`Transpose_684`	[1,192,7,7]
`Transpose_732`	[1,192,7,7]

4.1.3 Analysis.

As mentioned before, a different version of ShuffleNet V2 was already supported in MACE, but it was only for the CPU configuration. It could be that case that MACE developers found the CPU more efficient due to the better support of operators. The ShuffleNet V2 paper did experimental analysis of the model on a Qualcomm CPU, not GPU [3]. Thus, this was a first attempt at implementing ShuffleNet for mobile GPU. Further experiments on other devices should be done to know if the GPU can close the gap, but for now, our analysis of if GPU is actually slower or if our algorithm is not optimal is inconclusive. We cannot know for sure until we know the microarchtiectural details

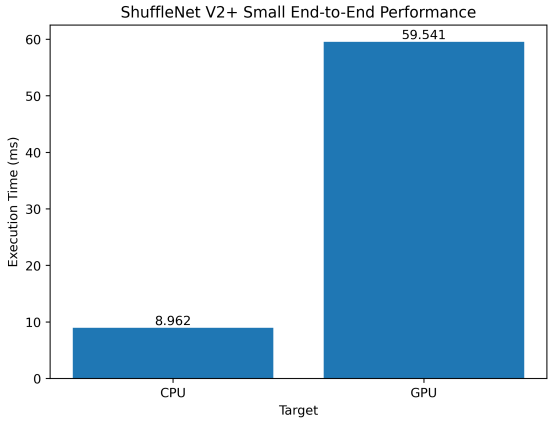

Figure 6. ShuffleNet V2+ Small End-to-End Performance

of the Ardreno 618 GPU to compare the peak memory bandwidth against the CPU bandwidth. Even though CPU and GPU share the main memory, the CPU for channel shuffle is single threaded, but most likely vectorized. On GPU, OpenCL manages how many threads or work groups, so it could be that GPU execution and memory units are not saturated enough based on the operator inputs to deliver peak performance.

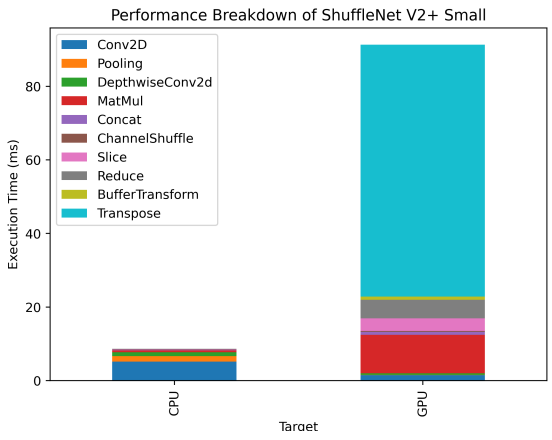

Figure 7. ShuffleNet V2+ Small Breakdown

4.2 RegNet (200M)

RegNet (200M) calls group convolution with the specific parameters shown in Table 2. In total, there are 13 grouped convolutions. And we see that only the 3×3 kernel is called. Thus we implemented and optimized CPU and GPU microkernels for grouped convolution, with also a generic grouped convolution as a fallback for any other model that doesn't call a 3×3 kernel.

For the implementation of group convolution, CPU and GPU were unimplemented in the MACE framework. Thus, we implemented and optimized CPU and GPU kernels, with a focus on GPU. Both implementations derived microkernels from regular convolution and added an extra for loop to iterate over the groups. In terms of loop ordering, we found that putting in the order of batch, group, output channel, input channel, height, width makes the most intuitive sense (this is an example given for NCHW case). Trying other orderings give sub-optimal performance. Groups should be place right after batch, since data locality can be benefited: there is no data dependency between the batch and group dimension.

Thus, for CPU implementation, we implemented two microkernels in NEON intrinsics: `GroupConv2dK3x3S1` and `GroupConv2dK3x3S2`. The only difference between the two are the strides. We took the regular Conv2d 3×3 kernels and added an extra for loop to iterate over groups. We also had to change the ThreadPool which was originally parallelizing over two dimensions: the batch and output channel dimension. We changed this ThreadPool dimension to include the groups as well. We also experimented with just two dimensions over batch and group dimensions, but we found out that doing three dimensions gives a slightly better runtime. Also, under assumption of batch size of 1, we are effectively performing threading over two dimensions, not three.

For GPU implementation, we do the same and just simply modify a regular convolution 3×3 kernel to iterate over groups. It's very simple implementation and gives us good performance.

Note that for the GPU configuration of RegNet, only the last layer, GEMM, fallsback to the CPU. We suspect that it fallsback to the CPU because it is the last layer and needs to serve the results to CPU anyways. But its performance is very low compared against CPU configuration. We suspect that it maybe due to the data format and a microkernel not being optimized. The operation that's happening is not really matrix multiplication or GEMM, rather matrix-vector multiplication or GEMV since we are still under the assumption of batch size of 1.

4.2.1 Grouped Convolution Performance.

Figure 8 shows the op performance of group convolution over the overall runtime of RegNet execution time for CPU and GPU configuration. We see that we indeed get a speedup of 5.84× when running on the GPU with a runtime of 461 microseconds. The operator performance on GPU is pretty good when compared against CPU. It's also much more faster than regular convolutions as we will discuss in the next section. To justify this performance, we can compare against MACE's depth-wise convolution operator, and see that it has similar throughput and execution time, since depth-wise is a special case of group convolution where number of groups equals to the number of input channels.

Table 2. RegNet (200M) Group Convolution Parameters

input	output	weight	bias	kernel	stride	padding	groups
1,24,112,112	1,24,56,56	24,8,3,3	24	3,3	2,2	1,1,1,1	3
1,56,56,56	1,56,28,28	56,8,3,3	56	3,3	2,2	1,1,1,1	7
1,152,28,28	1,152,14,14	152,8,3,3	152	3,3	2,2	1,1,1,1	19
1,152,14,14	1,152,14,14	152,8,3,3	152	3,3	1,1	1,1,1,1	19
1,368,14,14	1,368,7,7	368,8,3,3	368	3,3	2,2	1,1,1,1	46
1,368,7,7	1,368,7,7	368,8,3,3	368	3,3	1,1	1,1,1,1	46

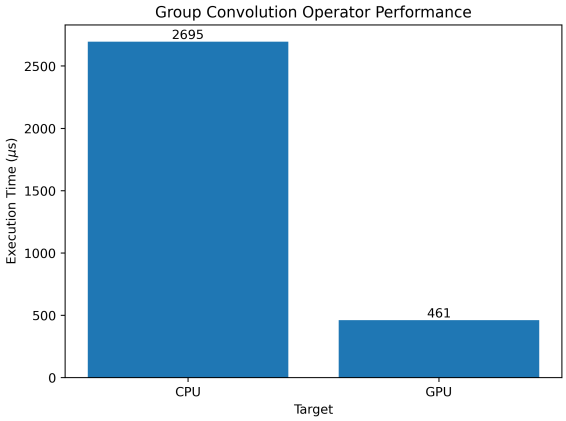

Figure 8. Group Convolution Op Performance

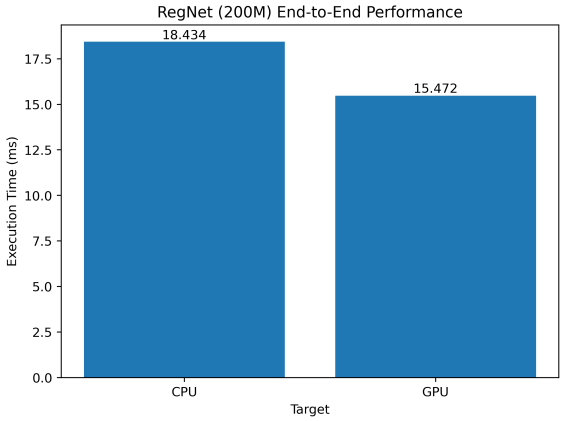

Figure 9. RegNet (200M) End-to-End Performance

4.2.2 End-to-End Performance. For GPU configuration of RegNet, only the last MatMul (GEMM) fallsback to CPU. Thus, the GPU configuration is pretty optimal in terms of performing its computations on the GPU. We get a speedup of 1.2× over the CPU configuration with an execution time of 15.5 ms. This equates to a 65 FPS throughput, making it good for real-time video applications. This is an 11 FPS improvement over the CPU configuration that can achieve 54 FPS. This is very minimal, but we find that when breaking down the performance op-by-op, MatMul (GEMM) takes the most time because it falls back to CPU as shown in Figure 10. If we can somewhat optimize MatMul on CPU or force it to run on GPU, we could potentially see much more speedups, equating to 8.9 ms or a 2.1× speedup. That equates to 112 FPS, and is beyond real-time for video applications. As mentioned before, the sub-optimal performance was also shown in ShuffleNet V2+, so further work needs to be done on how to improve the performance of the overall model on a GPU configuration.

4.2.3 Analysis. We see that our group convolution performance is ideal. But there is still more room to improve, because when observing the MACs, it doesn't quite reach the throughput compared against a regular convolution. Most of that is due to under-utilization of compute units are the convolution is split upon groups and smaller computations are done on small partitions of data. Thus, the compute units are not as saturated and exploiting cache and data reuse compared against regular convolution.

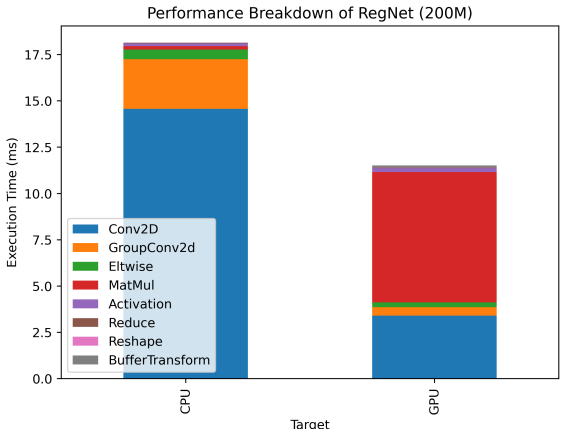

Figure 10. RegNet (200M) Breakdown

5 Challenges and Lessons Learned

Some challenges included getting the Android development setup correctly for productive development and debugging. We learned that early on, GDB is near impossible, unless the phone is unlocked or rooted. This can cause risks towards data loss or usability of the smartphone, so we avoided it and used log and print statements instead. The devices we used were production devices, so we were limited bypassing the security measures on the device. Unfortunately, this could mean there could still exists some bugs that are unseen by the debugger or missed by us. Fortunately, from what we tested, there seems to exist no bugs for the models we optimized and implemented over.

Other challenges included vague knowledge about how Ardreno GPUs work and trying to optimize performance for their microarchitecture. This is in contrast to CUDA devices, where GPUs released from NVIDIA usually have a whitepaper architecture report associated with them, listing out how many Streaming Multiprocessors (SMs) they have, how large the register file is, what accelerators are present (i.e. TensorCores), and more. Also, because this library is meant for multiple generations of devices supported under Android, we cannot necessarily optimize performance for one device without creating larger code size. For example, some quick results on a Google Pixel 7 showed that the performance is actually slower on GPU compared to the XiaoMi device we used. The Pixel 7 uses a ARM Mali GPU in contrast to a Qualcomm Ardreno GPU. The Pixel 7 is two generations newer than the XiaoMi, but because of the different GPUs, we suspect there might be something different in the GPU's microarchitecture to give us the suboptimal performance.

6 Future Work

Although end-to-end performance of the ShuffleNet V2+ model could not be fully accelerated to its full potential on the GPU, we wish to use the implementation and integrate it to another framework, CoCoPIE XGen. CoCoPIE XGen is also another DNN framework providing fullstack optimizations to DNNs for mobile devices, such as pruning, quantization, and compiler optimizations [2].

Based on the results, there is actually probably more room to optimize. Although the speedups from CPU to GPU are promising, the throughput or multiply-accumulates (MACs) doesn't quite reach as high as the ones with regular convolutions. There are also many issues regarding to the performance of group convolutions in PyTorch and other frameworks. The sole reason in the PyTorch case is because it is calling cuDNN kernels, making the performance issue on NVIDIA's end. But in general, GPUs are treated as one large SIMD unit, where only one kernel is run at a time. In the case of group convolution, it is optimal for multiple kernels to run concurrently per group. But because the individual grouped kernels are relatively small sized, they do not saturate the execution units enough to give peak performance.

7 Artifact Evaluation

To reproduce the findings in this paper, please head over to GitHub and clone the repository at https://github.com/briancpark/csc766-project. There are detailed instructions on how to install and setup the environment, as well as how to reproduce the results shown in this report.

Acknowledgments

This project was done under guidance of Dr. Xipeng Shen's course: Computer Science 766: Code Optimizations of Scalar and Parallel Programs. Android phone that was benchmarked and developed in this report was graciously lended by him. Also thanks to Dr. Bin Ren and Jiexiong Guan for giving guidance and suggestion of understanding the performance on mobile devices and the MACE framework.

References

[1] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. 2012. ImageNet Classification with Deep Convolutional Neural Networks. In *Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 1* (Lake Tahoe, Nevada) *(NIPS'12)*. Curran Associates Inc., Red Hook, NY, USA, 1097–1105.

[2] Xiaofeng Li, Bin Ren, Xipeng Shen, and Yanzhi Wang. 2022. CoCoPIE XGen: A Full-Stack AI-Oriented Optimizing Framework. https://doi.org/10.48550/ARXIV.2206.10620

[3] Ningning Ma, Xiangyu Zhang, Hai-Tao Zheng, and Jian Sun. 2018. ShuffleNet V2: Practical Guidelines for Efficient CNN Architecture Design. *CoRR* abs/1807.11164 (2018). arXiv:1807.11164 http://arxiv.org/abs/1807.11164

[4] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg, and Li Fei-Fei. 2015. ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision (IJCV)* 115, 3 (2015), 211–252. https://doi.org/10.1007/s11263-015-0816-y

[5] Jing Xu, Yu Pan, Xinglin Pan, Steven Hoi, Zhang Yi, and Zenglin Xu. 2022. RegNet: Self-Regulated Network for Image Classification. *IEEE Transactions on Neural Networks and Learning Systems* (2022), 1–6. https://doi.org/10.1109/TNNLS.2022.3158966

[6] Xiangyu Zhang, Xinyu Zhou, Mengxiao Lin, and Jian Sun. 2017. ShuffleNet: An Extremely Efficient Convolutional Neural Network for Mobile Devices. *CoRR* abs/1707.01083 (2017). arXiv:1707.01083 http://arxiv.org/abs/1707.01083

Received 1 May 2023