THOMAS: Tuning Highly-efficient Offline Multi-layer-NN on Android SoCs

Majeed Thaika, Anshu Aviral
Department of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213
{mthaika,aanshu}@andrew.cmu.edu

1 Introduction

In recent years, mobile devices have become powerful and have relatively high storage capacities - this has paved the way for mobile devices to support previously infeasible resource-intensive deep neural networks. There are many ML applications on mobile that can benefit from these DNN computations, including computer vision (object recognition and image generation), speech (mobile assistants and voice recognition), and reinforcement learning (mobile games). Nowadays, many popular ML frameworks offer off-device network training, most commonly utilizing the cloud, and then allow you to perform inference using the trained model on your mobile device. These frameworks usually provide CPU-only mobile support and focus on compressing networks so that they can provide compatibility with a wider variety of phones. Though we cede compatibility, we believe that leveraging integrated GPUs in the mobile devices can allow us to perform faster inference on deeper networks. Additionally, we may also be able to train or tune these models on the mobile device itself.

The primary approach behind machine learning solutions involve gathering data, using that data to train a model and then using the model to make predictions. With sensors like microphones and accelerometers in our phones, sampling thousands of data points per second and cameras capturing millions of pixels at several frames per second, there is no dearth of data required for training an AI agent. Training and predicting using a neural network, on the other hand, is a compute intensive task and is difficult to perform without parallelizing the tasks.

Successfully achieving the required level of parallelism necessitates powerful GPUs on mobile SOCs and a programming platform to leverage this massive computing power. One such example is the Nvidia mobile processor called Tegra K1, which has been incorporated into Nexus 9 and Google Tango that contains 192 GPU cores and conforms to the CUDA specification. Even with such specifications, a typical machine learning model will require days to train from scratch. On the other hand, inference, which is tantamount to a matrix multiplication, is a more viable task to parallelize.

1.1 Problem Statement

We primarily tackle the problem of parallelizing inference of convolutional neural network on mobile SOCs. Though easier than training a network, this is still a challenging task due to high memory capacity required both for storing the parameters and the activation values during the forward pass. The processing speed of mobile devices still remain in MHz while a typical laptop can clock up to 2 GHz, which means that performing computations on mobile phones is around 4x slower. Moreover, minimizing energy consumption remains an integral challenge since the battery life of mobile phones have direct correlation with user experience. We write code to optimize for memory performance, kernel performance and energy consumption.

Our optimization task is made more complicated by requiring our kernels to be able to generalize well to a wide range of features, parameters, and configurations, whilst under resource constraints.

Even within the same network architecture and hardware device, two adjacent kernels will have very different characteristics. Therefore, we try to perform OpenCL optimizations that should generalize well to layers in popular convolutional architectures.

We aim to provide a framework that will make it possible to fine-tune last layers of deep convolutional networks. This allows users to personalize pretrained networks offline using on-device user data with the added security benefit of never exposing their personalized neural network or data. We will also add support for end-to-end training for smaller CNNs.

2 Related work

Most popular deep learning frameworks are usually focused on improving compatibility for a wide range of devices. They rarely focus on optimizing for certain devices, especially if the device isn't already widely adopted by the community. Since deploying deep learning models on mobile devices is still in its infancy, there is minimal support for mobile GPUs (let alone CPUs). However, over the past few years, therer have been growing interest from the open source community to add mobile compatibility and optimizations for popular deep learning frameworks.

2.1 Tensorflow Lite (Google)

An mobile extension of TensorFlow, TensorFlow Lite translates your ML models into low-latency, small versions of itself. These lightweight models are decompressed-on-the-fly and can be executed quickly on your mobile devices. It's still early in its development cycle and has no GPU support so far.

2.2 MACE (Xiaomi)

MACE is a deep learning inference framework optimized for heterogeneous Android architectures. It supports Android OpenCL GPU inference, but is the source code is still proprietary.

2.3 Core ML (Apple)

Apple developed the Core ML framework to be used for deploying ML algorithms on their proprietary devices. Core ML allows model conversion from frameworks like TensorFlow Lite and Caffe2. It has GPU compatibility, but community consensus is that it has pretty poor performance to date, and doesn't support training on device yet.

2.4 QNNPACK (PyTorch)

QNNPACK (Quantized Neural Networks PACKage) is a mobile CPU-optimized library for NN inference, that trades off precision for higher performance. QNNPACK supports common NN operations on compressed 8-bit tensors. QNNPACK is integrated into PyTorch 1.0 and represented using Caffe2 graphs.

2.5 Fritz.ai

Fritz.ai provides an end-to-end solution that allows developers to convert and deploy ML models incorporated into your mobile apps.

3 System Specification

3.1 Hardware Specification

We conduct our experiments primarily on the Samsung Galaxy S8. The device has an Octa-core Kryo CPU and an integrated Snapdragon Adreno 540 GPU. Qualcomm's Snapdragon is one of the most widely used mobile system-on-chip in Android today. Therefore, our system can be run on many recent Android devices with little modification, but you may get different results due to the heterogeneity of these GPUs.

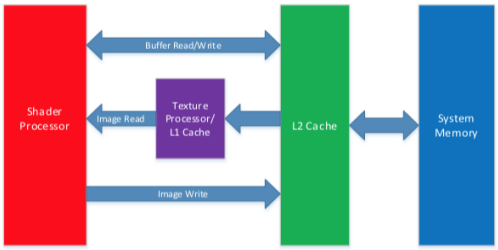

Figure 1: Architecture of Adreno 540 GPU

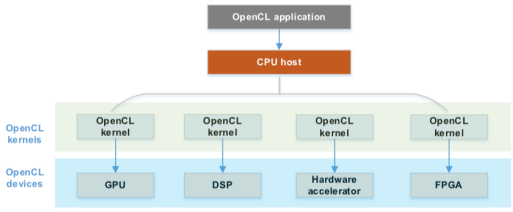

Figure 2: A heterogeneous system that supports OpenCL

Adreno 540 has an on-chip memory of 1024 KB, clocks 710 MHz, performs 567 GFLOPS and supports OpenCL API version 2.0, though we use OpenCL 1.1 for our experiments. A high level overview of Adreno 540 architecture that supports OpenCL is shown in Figure 1.

Adreno 540 currently supports up to 16 waves (equivalent to CUDA warps), with each wave supporting 64 fibers (equivalent to CUDA threads). Thus, for each kernel, the maximum workgroup size (equivalent to CUDA blocks) is the product of the maximum allowed number of waves and the wave fiber size (maximum workgroup size of up to 1024 fibers).

3.2 Software Specification

We use the Android NDK (Native Development Kit), a toolset for android app development, which allows us to code parts of our app in native languages like C and C++. This is particularly helpful as it enables us to use the OpenCL C++ libraries to perform-data parallel computations. A typical OpenCL based system, as shown in Figure 2, consists of a host CPU, multiple OpenCL devices like GPU, FPGAs and kernel codes that are compiled and loaded by the host to OpenCL devices for execution.

We extend the DeepCL library [1] which is a relatively small library for training deep convolutional neural networks on OpenCL-enabled machines. We also draw inspiration from TrasferCL [2], a framework which added Android support to DeepCL and offered some GPU support with lots of room for improvement.

4 Proposed Optimizations

To achieve maximal speedup, we want to optimally utilize the mobile phone's computing power and memory bandwidth. To do this, we can consider different layers of OpenCL optimization.

4.1 Application Layer

Due to hardware differences between different OpenCL-enabled devices, OpenCL does not have great performance portability. Thus, OpenCL applications that have been optimized on other platforms, like the DeepCL library being optimized on dedicated GPUs, usually perform poorly on smaller integrated GPUs like the Adreno 540. We also have to make greater resource trade-offs since we have less compute and storage capacity as compared to discrete GPUs.

When porting code optimized on different architectures to integrated mobile GPUs, two important factors we must consider are the input data size (network input and architecture) and arithmetic intensity. GPUs have high computation power compared to CPUs but can still be prone to poor speedup if the application is inherently memory-bound.

When porting over CPU-optimized code, we may combine multiple CPU module into one OpenCL kernel so that we reduce our accrued memory communication footprint. We may also split up a complex module into many smaller OpenCL kernels, which may give us less complex but more parallel efficient code.

4.2 OpenCL API Layer

Instead of compiling the kernels for every layer using *clCreateProgramWithSource*, we instead call *clCreateProgramWithBinary* the first time and load the saved binary file in each subsequent call. This may not affect kernel execution time, but might help reduce the total program execution time, especially if we have many repeating NN layers, like we do in VGGNet or ResNet. We build binaries at the start of training or prediction rather than load existing binary code because it can only be used with the specific architecture (Adreno A540) for which it was compiled.

Another optimization we tried to reduce total program execution time rather than performing kernel optimizations was to avoid creating or releasing memory objects between *clEnqueueNDRangeKernel* calls, since the amount of memory requested is inversely proportional to the kernel execution time. Instead, it is encouraged to reuse memory objects in OpenCL instead creating new objects. The CPU should be doing little/no memory manipulation when launching GPU kernels so as to reduce GPU execution stalls. Since our proposed frameworks runs CNNs that don't pass in or modify data between layers, we sacrifice some functionality for faster program execution time.

We use the Android ION memory allocator, *clCreateBuffer*, which creates OpenCL memory objects with ION (zero memory copy) pointers instead of allocating additional memory and copying over the objects. We also set optimization flags, such as read-only memory, to improve memory load times.

We also tried to avoid blocking memory API calls where possible. Blocking calls stall the CPU while waiting for the GPU to finish. It also stalls the GPU before the next clEnqueueNDRangeKernel call. However, since the input of the next layer kernel depends on the output of the previous layer kernel, there may limited scope for optimization here.

4.3 Kernel Layer

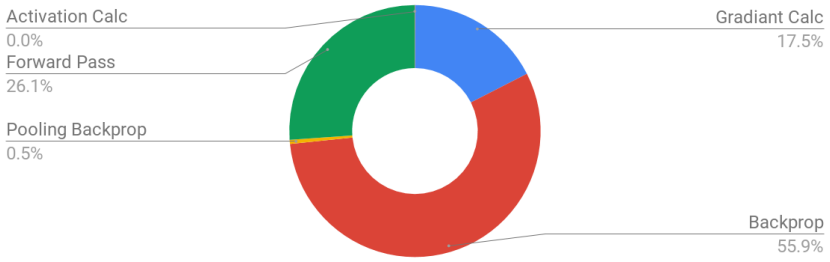

Figure 3: Time taken per Kernel for training complete LeNet on MNIST

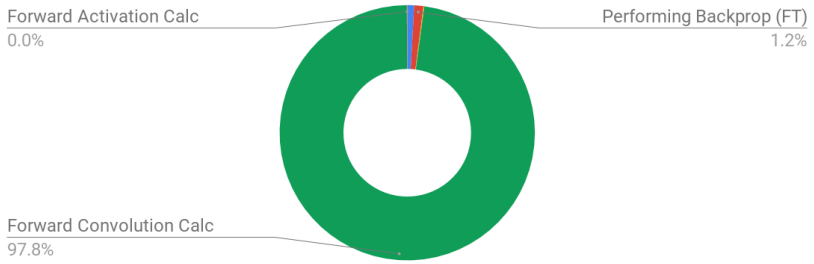

Figure 4: Time taken per Kernel for fine-tuning last layer of LeNet on MNIST

Since we limit our optimization scope to fine-tuning the neural network's last layer rather than the full network, we observed that forward convolution is actually the major bottleneck, as seen in Figures 3 to 8, and so we decided to optimize the forward pass rather than backpropagation. In our analysis, we may provide the execution time for both forward and backward pass if we added a optimization to those relevant kernels, but the majority of our focus was on the forward pass.

For each forward CNN layer, the convolutional module, the max pooling module, normalization module, and activation module were all given separate kernels in the DeepCL framework. This offers

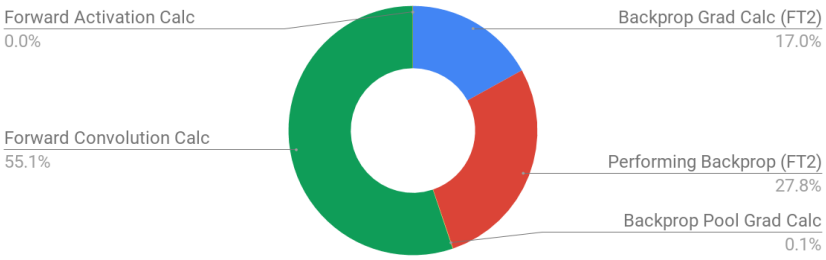

Figure 5: Time taken per Kernel for fine tuning final two layers of LeNet on MNIST

great modularity and easy customization, but given we have only one GPU in most mobile devices, synchronization between these layers could prove to be very costly. Instead, we chose to combine these modules in each CNN layer into one kernel. This means that we have to offer much more functionality and flexibility when building this kernel. This will add to the kernel complexity and will exhibit different characteristics as compared to the four separate kernels, so we will need to explore different kernel optimizations than those used by DeepCL.

With our proposed configuration, we have to reduce kernel complexity as much as possible. Roughly speaking, the more the kernel complexity, the larger the register footprints, so we can fit fewer active waves. This ultimately results in a smaller maximum workgroup size. However, on the flip side, if a kernel runs for too long a period, it triggers an alert that causes the Adreno GPU to reset - so a balance between kernel complexity and execution time should be achieved. It is recommended that each kernel execution time be in the order of tens of milliseconds.

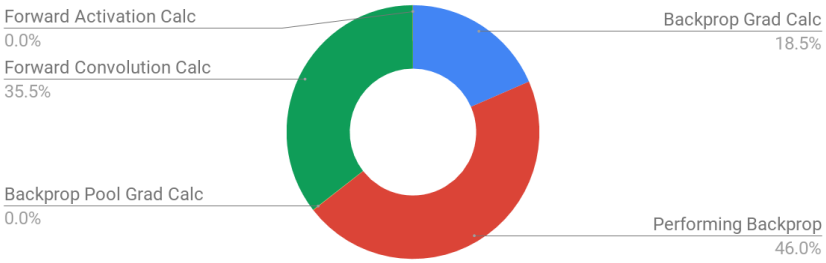

Figure 6: Time taken per Kernel for training complete AlexNet on ImageNet

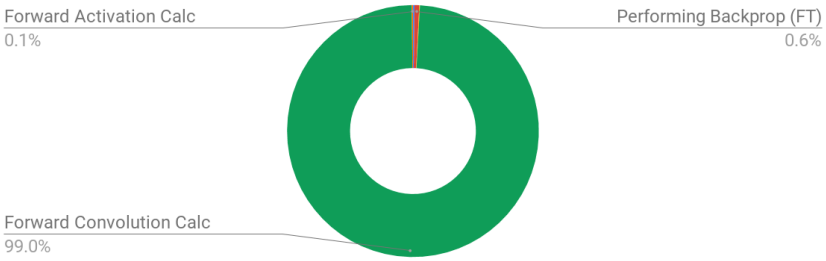

Figure 7: Time taken per Kernel for fine-tuning last layer of AlexNet on ImageNet

Some factors that contribute to higher register footprints are high-precision math functions, large private memory, large shared local memory, loop unrolling, control flow, and barriers. We tried using fast math, which gives up some numerical precision for higher performance but might not satisfy IEEE 754 floating-point requirements. We minimized the amount of private memory by precomputing non-input dependent variables before defining the kernel. We also kept the number of divergent control flows in the kernel to a minimum - we try to offer different functionality . We experiment with loop unrolling inside our kernel which increases kernel complexity (possibly fewer active waves) to hide memory latency.

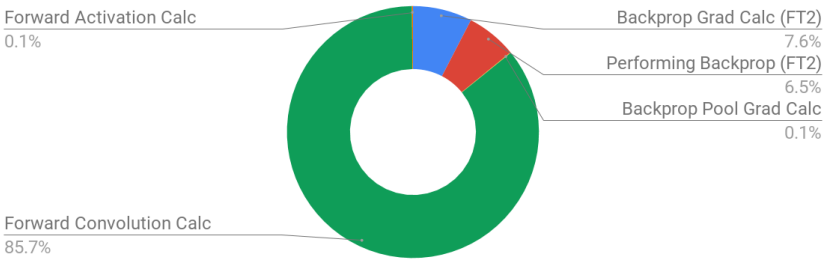

Figure 8: Time taken per Kernel for fine tuning final two layers of AlexNet on ImageNet

We decided to remove DeepCL's shared local memory feature in our forward CNN layer kernel because we now would need larger local memories (to incorporate max pooling). This lead to lots of cache conflict misses, smaller work group size, and further complicates the code. This also removed the need for costly barriers in the forward convolutions, which were mainly used to synchronize local memory access between fibers of the same workgroup.

We try to improve our GPU ALU performance by applying the aforementioned complexity reduction techniques and speeding up calculations by allowing the compiler to leverage fast relaxed math for networks where we can tolerate some loss in precision. We also explored using native math, where calculations are done directly in hardware so have even better performance than fast relaxed math, but since most of our calculations were just addition and multiplications, there wasn't much room for improvement. Another ALU optimization we tried was using 16-bit floating points instead of 32-bit floating points inside our kernel. This again does the familiar precision tradeoff for better GPU performance. Adreno GPUs have dedicated hardware to accelerate *half* (16-bit FP format) data type calculations. The GFlops of half ALUs is almost twice of full ALUs.

To alleviate memory bound problems so that the GPU doesn't stall too long, we try to optimize memory access by vectorizing load/store using *float4* or *half4*. We also used shorter data types for data transferred between GPU and memory so that we reduce memory bus traffic, and hopefully reduce GPU stalls. We thought about using OpenCL's image buffer to store the input data to our DNNs in the L1 image cache rather than using the L2 object buffer as seen in Figure 2. We however ultimately decided against using this approach primarily because we don't need any of the OpenCL image optimizations and don't want to risk L1 cache thrashing.

5 Baseline

Though we didn't perform in-depth comparison between our implementation and the framework provided by Tensorflow Lite and PyTorch's QNNPack, we did observe that the forward pass of their sequential MNIST implementation both took an average of 8-10ms per image on the S8's Snapdragon 835 CPU. Using our THOMAS framework, we took 22.8ms for inferring the labels of 128 images in parallel using the Adreno 540 GPU.

6 Experiment Setup

6.1 Comparing different architectures

For our experiments, we choose two well-known deep convolutional neural networks, LeNet and AlexNet. We benchmarked our optimizations using the MNIST and CIFAR-10 datasets on LeNet, and ImageNet on AlexNet.

6.2 Snapdragon Profiler

We used the Snapdragon Profiler to analyze Snapdragon-powered Android SoC performances through monitoring the CPU, GPU, memory, power, and thermal statistics over time. This helped us identify and rectify potential bottlenecks.

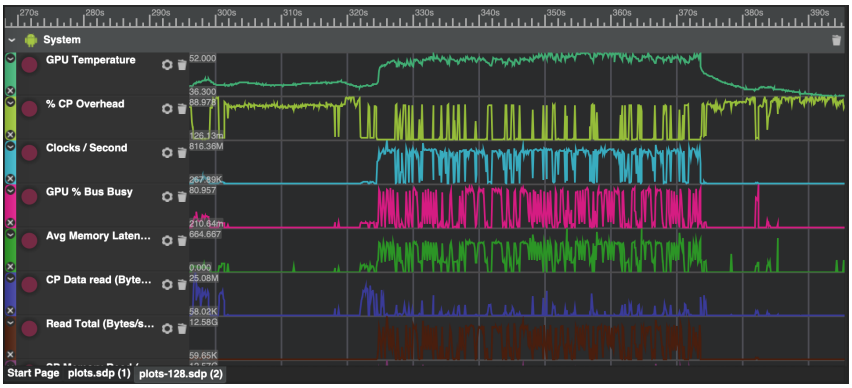

Figure 9: A Snapdragon run for LeNet on MNIST dataset for batchsize 128, showing important performance parameters like % CPU Overload and GPU % Bus Busy

6.3 OpenCL Kernel Profiler

The OpenCL Kernel profiler helps us analyze the duration of each kernel on the queue. In our case, the profiler helps us identify how long each kernel takes to execute to completion on the GPU. We can then identify performance bottlenecks and profile our optimizations.

7 Experiment Results and Discussion

For our three architecture-dataset pairs, we tried seeing what effect different batch sizes would have on the program execution time (see Figures 10 to 12). We found that the optimal batch size configurations for our framework are 128 images for LeNet-MNIST pair, 16 images for LeNet-CIFAR, and 8 images for AlexNet-ImageNet. Additionally, for LeNet we notice that there are lots of time being spent on non-GPU operations for smaller batches. When we dug into the Snapdragon Profiler, we saw that this was because each kernel was reading memory much less than its capacity as seen in Figure fig:batch-read. This suggests that the program is memory bound. However, we see almost no non-GPU execution time using AlexNet-ImageNet, suggesting these programs are more compute bound.

We notice that the time being spent on forward pass stays decreases as we increase batch sizes. Since we have no barriers or shared memory in our forward CNN layer, this process is memory bound when using smaller batches, but fully utilizes the maximum workgroup size as we increase batch size, giving us good performance.

Backpropagation execution time increases as we deviate batch size from optimal. For large batch sizes, this is likely due to cache thrashing from local memory and performance degradation from barriers waiting for stragglers. For smaller batch sizes, this is likely because we aren't fully utilizing the maximum workgroup size available to us.

Initially, we used non-blocking memory read, write, and mapping operations - especially since the input of traditional CNN layers are dependent on the output of the previous layer. We tried to loosen this restriction by using non-blocking writes because we reasoned that the network doesn't need to block twice while writing the last layer and reading the next. As seen in Figure 15, we get faster backpropagation times but surprisingly slower times for forward pass and non-GPU execution. A likely explanation is that we may get stragglers and have to eventually wait longer for the data to propagate forward. Since we have barriers in backward pass, being blocked fewer times may be beneficial.

We expected the fast relaxed math optimization to be faster than unoptimized 'slow' math case, but this was not what we observed. As seen in figure 16, there is negligible difference between the two execution times. This is likely due to our kernels mostly doing addition and multiplication calculations which are already quite fast and won't affect the runtime nearly as much as other factors.

When changing the forward pass kernel to use 16-bit floating point data type instead of its 32-bit counterpart and the complementing *half* functions, we notice a big speedup in the forward execution

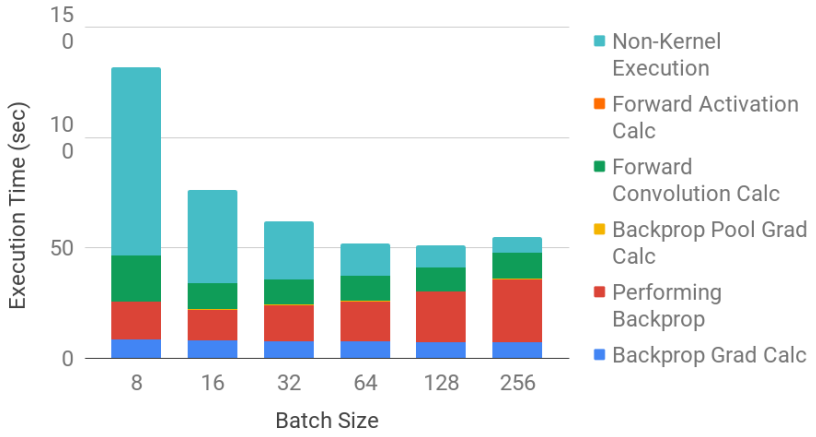

Figure 10: Program and Kernel execution time with increasing batch size for LeNet on MNIST

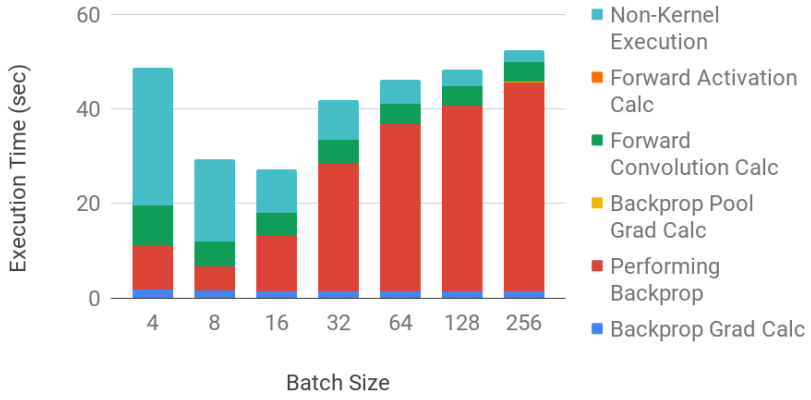

Figure 11: Program and Kernel execution time with increasing batch size for LeNet on CIFAR

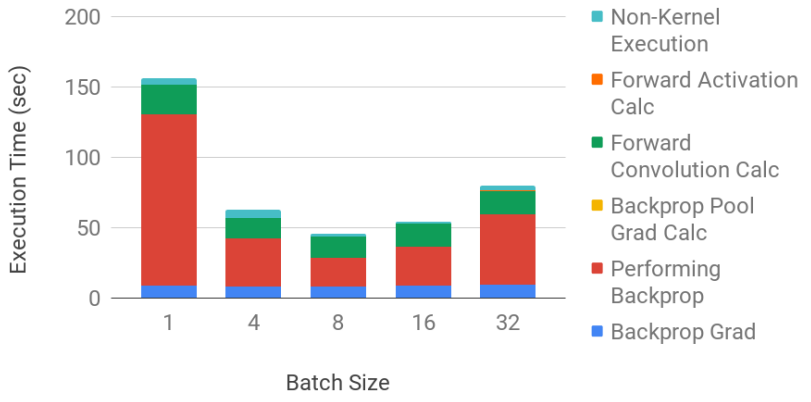

Figure 12: Program and Kernel execution time with increasing batch size for AlexNet on ImageNet

time. This is likely because the calculations are being done by the dedicated Adreno *half* hardware which runs at GFlops almost twice that of the 32-bit hardware.

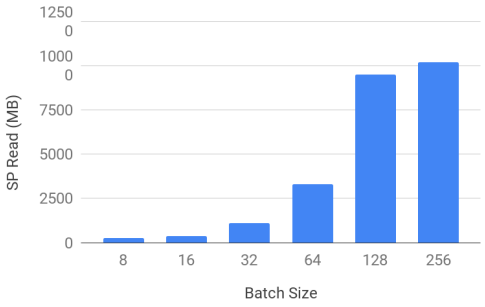

Figure 13: SP Read Sizes with increasing Batch Size (LeNet-MNIST)

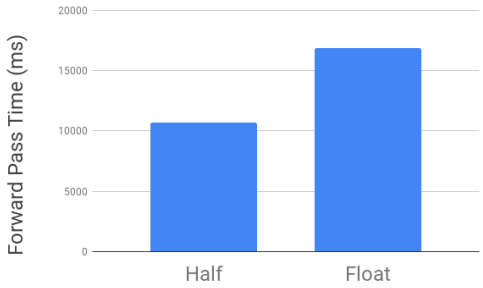

Figure 14: Comparison between using 32-bit floats vs 16-bit floats i.e. half (LeNet-MNIST)

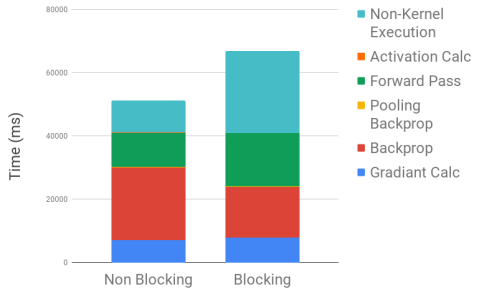

Figure 15: Comparison between blocking and non-blocking kernels (LeNet-MNIST)

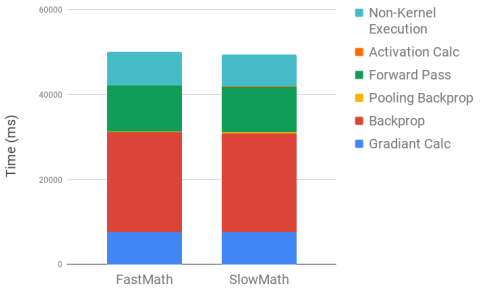

Figure 16: Comparison between fast relaxed math and slow math kernels (LeNet-MNIST)

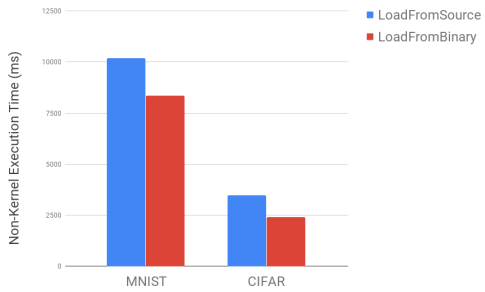

Figure 17: Comparison between Load from Source and Load from Binary for LeNet over MNIST and CIFAR datasets

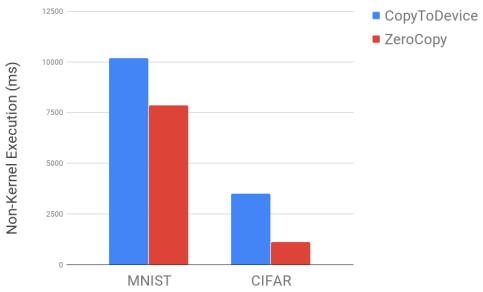

Figure 18: Comparison between Zero Copy and Copy to Device for LeNet over MNIST and CIFAR datasets

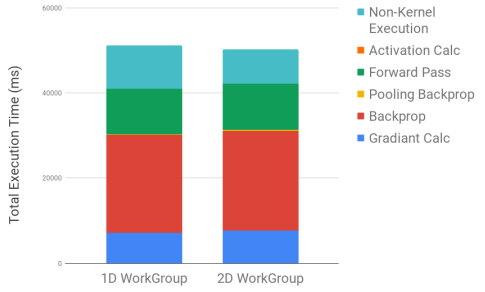

Figure 19: Comparison between 1D and 2D Work Group assignment of LeNet for MNIST dataset

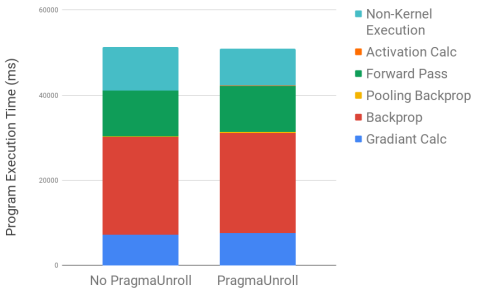

Figure 20: Comparison between with and without Pragma Unroll for LeNet on MNIST dataset

As expected we see that loading data from binary once is faster than building the source every time as seen in Figure 17. Zero copy mapping also performs better than copying from device since we avoid costly memory copy at the host side - this can be seen in Figure 18.

We expected 2D workgroup assignment to perform better than our original 1D assignment to leverage the locality of convolution operations. However, we notice negligible difference in Figure 21, probably because we didn't modify our access pattern to a blocked approach to actually leverage the cache.

We expect loop unrolling to hide memory latency within each kernel as it prefetches data, but we can see that this isn't actually the case in Figure 21. This is most likely because the compiler itself does loop unrolling optimizations where possible, and it could be the case that our *#pragma unroll* flag is redundant and/or not helpful.

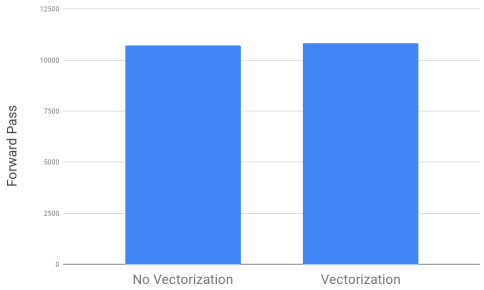

Figure 21: Comparison between with and without Vectorization for LeNet on MNIST dataset

Vectorized load/store in the forward layer gave negligible performance improvement (see Figure 21), probably because the programs weren't memory bound.

We also monitored the temperature and power of the Samsung S8 as we trained each of the networks (see Figures 23 to), since we don't want the mobile phones to reach very high temperatures and consume lots of power, because of possible hardware degradation (since mobile phones have poor ventilation and battery life). As expected, the larger network consumed most power and made the device the warmest amongst the three networks. However, our framework still posts much better thermal and power stats than the equivalent training task on the mobile CPU.

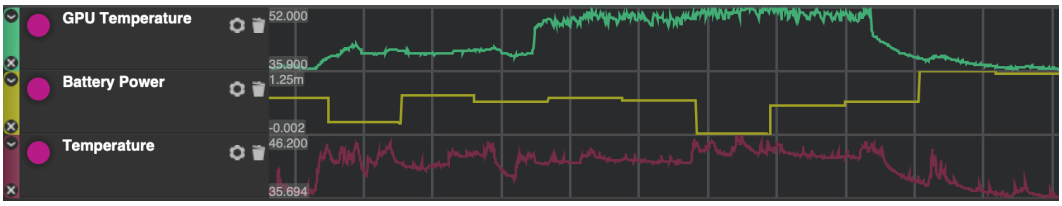

Figure 22: LeNet on MNIST Temperature and Power Statistics

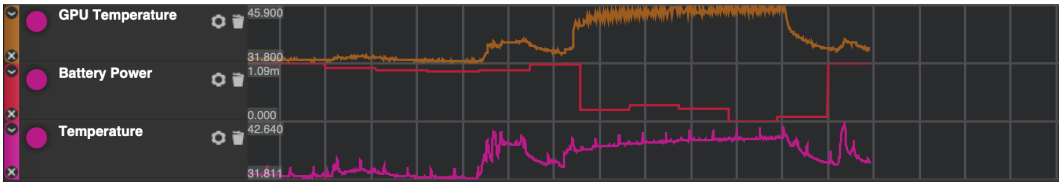

Figure 23: LeNet on CIFAR Temperature and Power Statistics

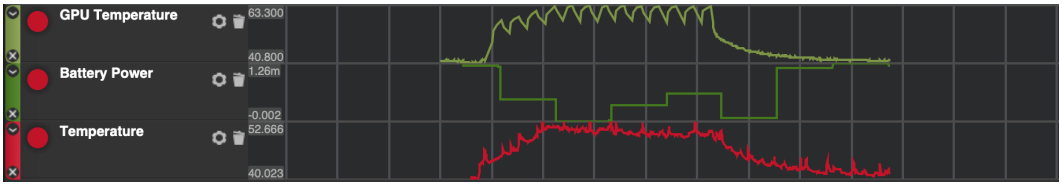

Figure 24: AlexNet on ImageNet Temperature and Power Statistics

References

[1] Hugh Perkins. Deepcl. https://github.com/hughperkins/DeepCL, 2018.

[2] Olivier Valery. Transfercl. https://github.com/hughperkins/TransferCL, 2018.