The performance optimization on TensorFlow framework on Mobile GPU devices using OpenCL

Wei Cheng
Department of Electrical and Electronic Engineering
The University of Hong Kong (HKU)
Hong Kong, China
u3512488@hku.hk

***Abstract* —-** The advancement of mobile computing technology and the recent progress in AI have driven the prosperity of edge computing, which means the computation used to happen in the cloud is now shifting to edge devices. Before the blossom of smart phones, mobile devices merely served as a communication medium; however, it's so powerful and energy efficient now, it's capable of operating intensive AI computation within a reasonable power budget. Yet, not all open-source AI frameworks in the market support AI training on mobile devices. In this paper, the feasibility of training a small AI task, the MNIST handwritten dataset, using Tensorflow framework on mobile CPU/GPU was demonstrated. To further optimize the Tensorflow framework performance on mobile devices. Benchmark programs were executed on mobile GPU to better understand the underlying architecture. Based on the benchmark results collected, GPU optimization techniques were applied to conquer the system bottleneck. As a result, the matrix multiplication task was accelerated by 2.16x times compared to the baseline performance.

Keywords: *OpenCL, mobile GPU, Tensorflow*

I. INTRODUCTION

A. Software — Tensorflow

Tensorflow is an open-source numerical computation library first introduced by Google in 2015. It's popular in both industry and research area because of its generic data flow programming model which makes it extensible to handle a wide range of neural network architecture. Among all other open-source machine learning frameworks, Tensorflow is chosen because of its popularity among the developer community.

B. Hardware — Mobile GPU

GPU was originally developed for better graphic display on desktop computers. Unlike CPU, GPU has more cores while each of them is less powerful and operates at lower speed. Nonetheless, packing massive amount of GPU cores on a chip gives great performance for graphics because it needs simple computation for each pixel and numerous pixels shall be processed for each frame, and several frames per second. The characteristics of GPU makes it naturally suitable for neural network AI task because of its highly parallelizable nature.

The computational power of mobile GPUs and desktop GPUs are at different level. Mobile GPUs are restricted by TDP (Thermal Design Power) since most mobile GPUs are packed with CPU into a SoC (System On Chip), and they have to share the TDP quota. Moreover, mobile GPUs have to share the last level memory with CPU [1] while desktop GPUs come with a piece of dedicated memory on chip separated from CPU.

C. Programming model — OpenCL

OpenCL is the industry driven SPMD (Single Program Multiple Data) programming model for GPU [2]. Unlike other mobile GPU programming framework such as RenderScript used in RSTensorflow [3], OpenCL exposed the underlying hardware to the developers which is more flexible and extensible.

II. LITERATURE REVIEW

Accelerating AI framework on mobile devices has been a popular research topic. In this section, related acceleration frameworks will be listed.

RSTensorflow [3] leveraged the RenderScript framework to accelerate the matrix multiplication and convolution operations on Android devices and achieved 3 times speedup in Google Inception_v3 model inference task. Qualcomm Snapdragon Neural Processing Engine (SNPE) is the official framework from Qualcomm supporting fast AI model inference on mobile devices using mobile GPU; however, the implementation isn't open source and the training feature isn't supported. Tensorflow Lite was released by Google in Nov 2017. It accelerates Tensorflow model inference process on mobile CPU. In detail, pre-fused activation, and quantized data were added to allow faster machine learning inference. Plus the AI model file is smaller by introducing a new format called "Flat Buffer", which is a new serialization library similar to the original one but without the need of parsing/ unpacking the text-based representation.

III. THEORETICAL PRINCIPLES

A. SPMD programming model

GPU is faster in some applications because of the parallel programming model. In OpenCL it's called SPMD, which means a group of work-items are executing the same instruction in a lock step with each other. Therefore, despite the fact that OpenCL compute units are slower than normal CPU cores, the concept of latency hiding enables GPU to achieve high throughput. By the time a GPU core is waiting for a memory access to the memory system, it's able to switch to another thread and executes a few more ALU instructions there until it encounters another memory instruction. As the memory system returns the requested data to the GPU core, it quickly switches back to the first thread and moves on to the next instruction. In that case, a GPU be kept busy all the time and the latency can be hidden. Nonetheless, it's possible for a GPU core to suffer from low throughput because the kernel itself is memory

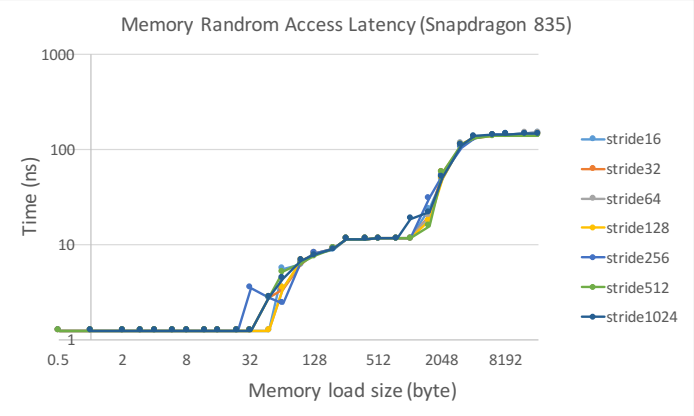

Figure 1. is the memory random access latency for Snapdragon 835 platform. X-axis is the memory load size in byte. Y-axis is the memory random access time in microsecond (us). Multiple data points on the right show the stride size for random memory access. Stride sizes from 16 bytes to 1024 bytes were performed in this experiment.

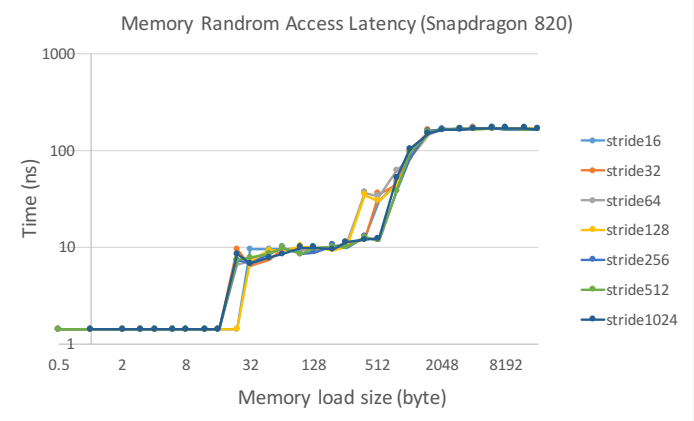

Figure 2. is the memory random access latency for Snapdragon 820 platform. X-axis is the memory load size in byte. Y-axis is the memory random access time in microsecond (us). Multiple data points on the right show the stride size for random memory access. Stride sizes from 16 bytes to 1024 bytes were performed in this experiment.

bounded or the memory access pattern isn't well supported by the underlying memory system.

B. Optimal memory access pattern

There're some optimal memory access pattern in GPU programming which best fit the underlying memory system. First, coalesced memory access refer to the capability of combining load store request from neighboring work-items. The Adreno 5xx series GPU supports coalesced load/store to local memory and coalesced load to global memory [1]. Secondly, vectorization refers to accessing memory in a vectorized way for a single work-item to better utilize the memory bandwidth. The best vectorization parameter is device dependent. Experiments profiling the best vectorization ratio on Adreno 540 GPU will be shown in this paper. Thirdly, it's a good practice to load or store a chunk of bytes from the memory, and load/store memory address should be 32 bites aligned [1]. Fourthly, local memory is shared with all work items within the same work-group. A kernel is faster with local memory because the access time is lower than global memory.

IV. METHOD OF INVESTIGATION

In order to apply the optimization technique to the system. Deep understand of the underly hardware is needed despite the fact that most mobile GPU architectures are proprietary and close source. By executing existing or self-designed benchmark programs on the mobile platform, information crucial for optimization can be revealed.

A. Benchmark — LMBench

LMBench [4] is a popular memory benchmark program testing the memory access latency of CPU. This benchmark is necessary because of the hardware architecture of mobile GPU. Unlike traditional desktop GPU with on-chip memory separated from the system RAM, mobile GPU has to share the last level memory with CPU. Therefore, by understanding the memory access latency of the system RAM, one could have a rough picture of memory latency in GPU. Which is a critical parameter when it comes to the optimization of GPU kernels.

The original software was written for UNIX system. It's cross-compiled to Android platform with Android NDK version 16 toolchain. The experiment was conducted on both Snapdragon 835 and Snapdragon 820 SoC, results are shown on Figure 1 and Figure 2 respectively.

From Figure 1 and Figure 2, the CPU L1 cache access time for S835 is ~1.2 ns, and S820 is ~1.4 ns. L2 cache access time for S835 is ~11 ns, and S820 is ~10 ns. The system RAM access time for S835 is ~145 ns, and S820 is ~170 ns. Due to the fact that the OpenCL global memory on mobile GPU is the system RAM. Conclusion can be made that the global memory in OpenCL memory model on mobile GPU has access time equals to ~170 ns and ~145 ns for S820 and S835 respectively.

B. Benchmark — MixBench

MixBench [5] is an OpenCL benchmark testing the relationship between three factors (throughput, memory bandwidth, operation intensity) on a GPU. As the operation intensity grows, the kernel moves from a memory bound kernel to a compute bound kernel. By profiling these factors, one can use these parameters to design a more efficient GPU kernel.

This experiment was conducted on both Snapdragon 835 SoC with Adreno 540 GPU and Snapdragon 820 SoC with Adreno 530 GPU. Integer operation, single precision floating point operation (FP32), and half precision floating point operation (FP16) were tested on both platforms.

The Adreno 540 performance of FP32 is shown in Figure 5, FP16 performance on Figure 6, Integer operation performance on Figure 7. The Adreno 530 performance of FP32 is shown in Figure 8, FP16 performance on Figure 9, Integer operation performance on Figure 10. As the operation intensity increases, the kernel is switching from a memory bound kernel to a compute bound kernel. From Figure 5 to 10, there's a discontinuous point (marked by red arrow) in each Figure, which represents a burst in both computation and memory throughput. This is the sweet region for a GPU kernel to yield maximum hardware utilization by latency hiding.

C. Benchmark — OpenCL memory bandwidth test

Inspired by the bandwidth tests in Qualcomm Adreno SDK, this benchmark was built from scratch to measure the memory transfer bandwidth between the host device and OpenCL devices. Data consumed by an OpenCL kernel should be first loaded into the GPU memory by the *`clEnqueueWriteBuffer`* function in the OpenCL

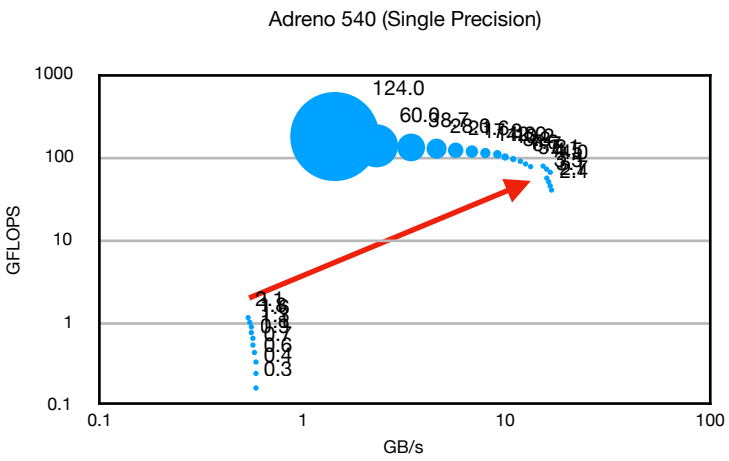

Figure 5. The performance of single precision floating point operation on Adreno 540 GPU. X-axis is the memory bandwidth in GB/s. The Y-axis is the throughput (GFLOPS). The diameter of the data point is the kernel operation intensity in FLOP/Byte. The number right next to the data point is the operation intensity.

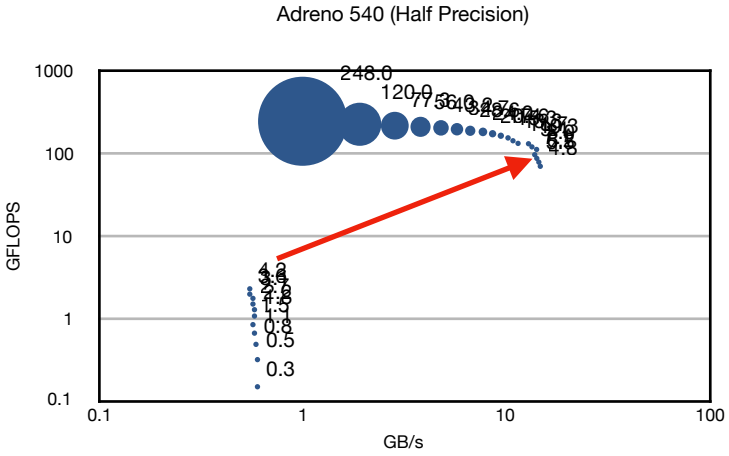

Figure 6. The performance of half precision floating point operation on Adreno 540 GPU. X-axis is the memory bandwidth in GB/s. The Y-axis is the throughput (GFLOPS). The diameter of the data point is the kernel operation intensity in FLOP/Byte. The number right next to the data point is the operation intensity.

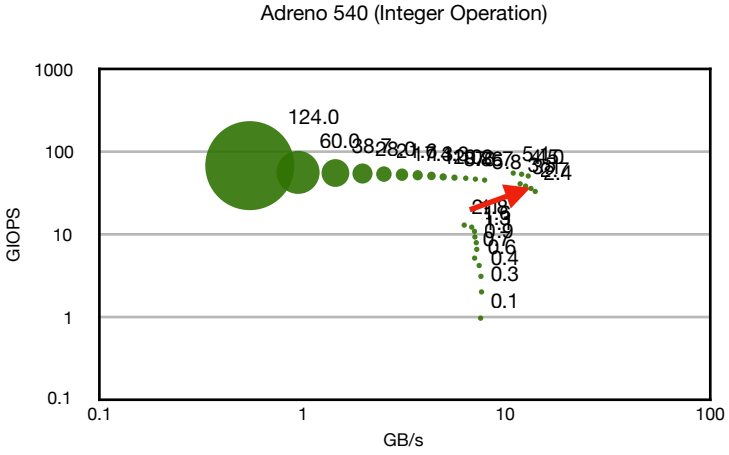

Figure 7. The performance of integer operation on Adreno 540 GPU. X-axis is the memory bandwidth in GB/s. The Y-axis is the throughput (GIOPS). The diameter of the data point is the kernel operation intensity in IOP/Byte. The number right next to the data point is the operation intensity.

programming model. Similarly, results computed by a kernel should be read back to host using the `*clEnqueueReadBuffer*` function. The data transfer time between GPU and CPU consists of a large portion of computational time. Thus, understanding such performance is important for optimization. Memory bandwidth was measured in the following scenarios. Data transfer from

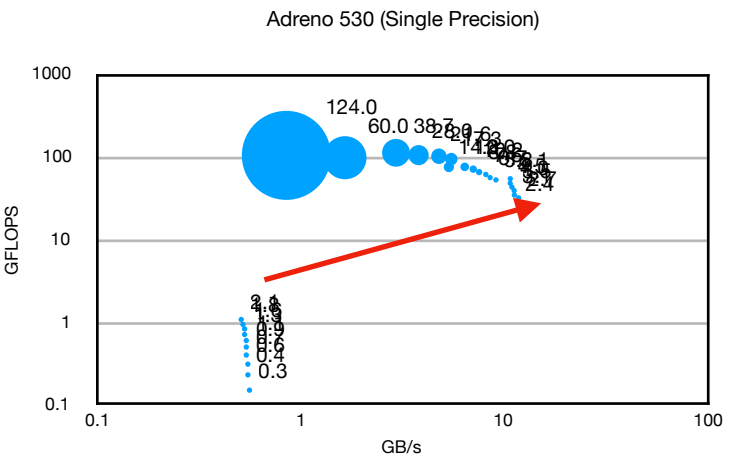

Figure 8. The performance of single precision floating point operation on Adreno 530 GPU.

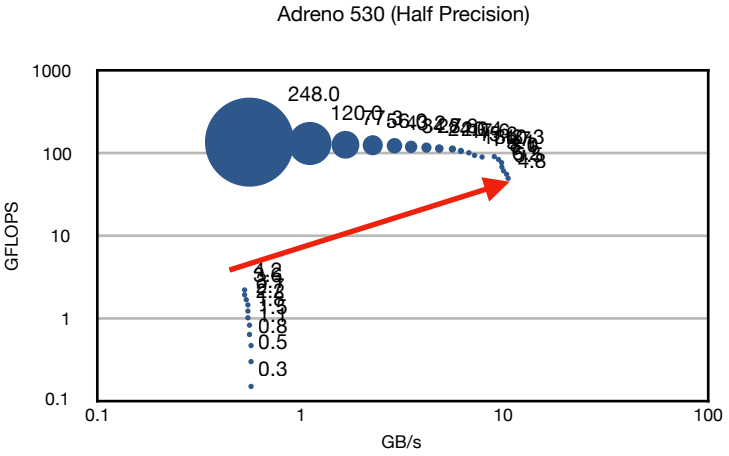

Figure 9. The performance of half precision floating point operation on Adreno 530 GPU.

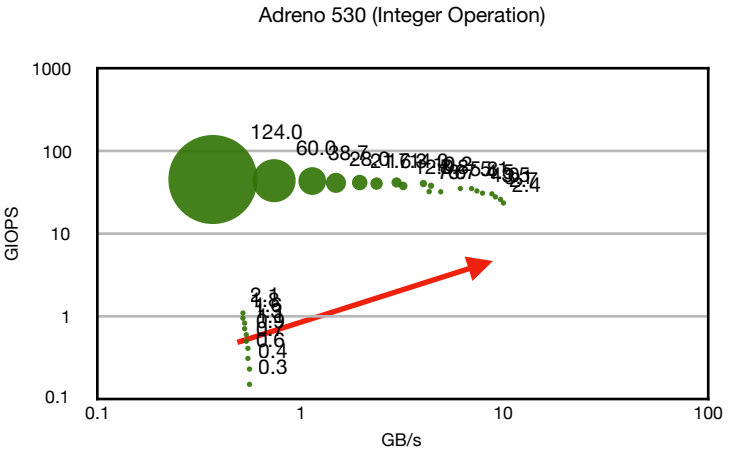

Figure 10. The performance of integer point operation on Adreno 530 GPU.

host to OpenCL devices, from OpenCL devices to host, and from an OpenCL device to another OpenCL device.

Unlike the traditional desktop GPUs, the global memory on mobile GPU is shared with system RAM. As a result, the obtained results should be the transfer bandwidth within the system RAM. Moreover, the OpenCL buffer memory and image memory are handled differently on Adreno GPU [1]. In this experiment, only OpenCL buffer memory object was tested. Results of Adreno 540 GPU are shown in Figure 3, and results of Adreno 530 GPU are shown in Figure 4. Notice the difference between Figure 3 and Figure 4, both Adreno 540 and Adreno 530 devices have similar host-to-device and device-to-host

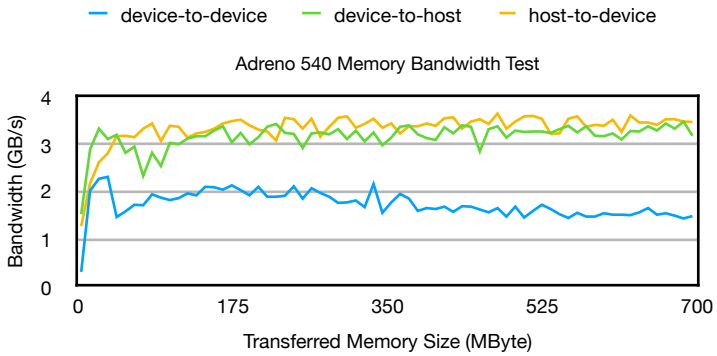

Figure 3. This chart shows the memory transfer bandwidth between host device and OpenCL device on Adreno 540 GPU. X-axis is the memory size being transferred in Mbyte. Y-axis is the measured bandwidth in GB/s.

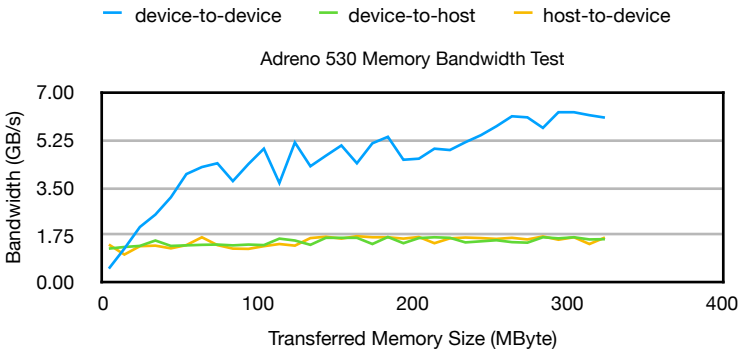

Figure 4. This chart shows the memory transfer bandwidth between host device and OpenCL device on Adreno 530 GPU. X-axis is the memory size being transferred in Mbyte. Y-axis is the measured bandwidth in GB/s.

memory transfer bandwidth. Expectedly, the device-to-device bandwidth on Adreno 540 GPU is half of those between host and device because the data should be read and sent back to OpenCL devices, the channel is shared so half of the bandwidth is reasonable. Unexpectedly, the device-to-device memory transfer bandwidth on Adreno 530 GPU is 2 to 3 times faster than host-to-device bandwidth.

V. EXPERIMENT RESULTS

A. Experiment — CLBlast evaluation

The CLBlast library is an open source OpenCL BLAS library [6]. It's designed to leverage the performance of various kinds of OpenCL devices ranging from desktop GPUs to mobile GPUs. The library consists of two parts, the BLAS library which provides basic library algebra operations, and a tuner that runs automated tests on an OpenCL devices and generates the a combination of parameters that gives the best performance. In this experiment, only the GEMM (GEneral Matrix-to-matrix Multiplication) functionality of the BLAS library was tested.

A.1. Untuned version

Notice that a database is embedded in the library to select the appropriate set of parameters for the BLAS OpenCL kernel at runtime. It first identifies the device name and the device vendor by the OpenCL `*clGetDeviceInfo*` function and uses the retuned value to select a set of parameters for that device. The default set of parameters for Adreno GPU is tuned for Adreno 330. The performance of the untuned version is shown in Figure 11.

A.2. Tuned version

As instructed by the CLBlast manual, tuning the performance for a new OpenCL device is needed to find the best set of parameters for the OpenCL kernel. An ideal set of parameters for Adreno 540 was obtained by running the tuner manually on the devices and the best set of parameters were added to the database. The matrix multiplication result of the tuned version CLBlast is shown in Figure 12.

A.3. Tensorflow overhead

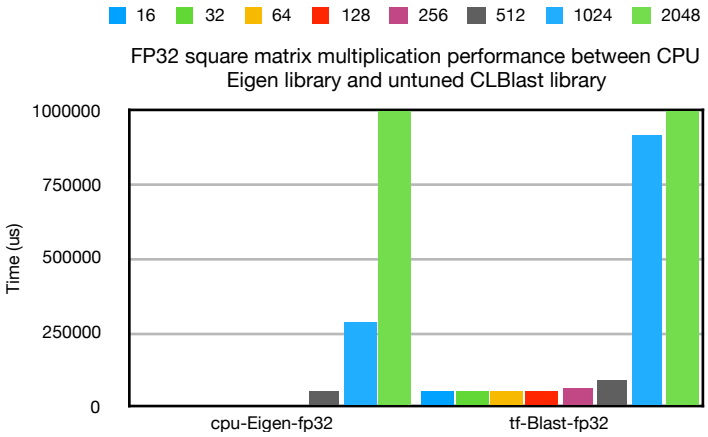

Figure 11. The FP32 square matrix multiplication performance between CPU Eigen library and untuned CLBlast library. Different colors show matrices of different size. Y-axis is the time needed in microsecond (us).

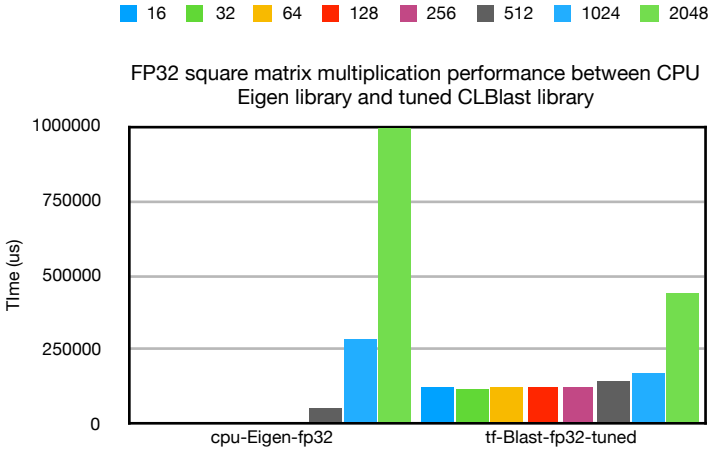

Figure 12. The FP32 square matrix multiplication performance between CPU Eigen library and tuned CLBlast library. Different colors show matrices of different size. Y-axis is the time needed in microsecond (us).

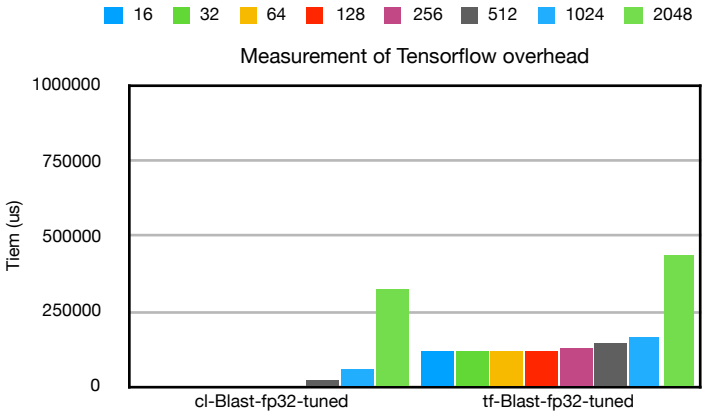

Figure 13. This is chart shows the Tensorflow overhead when incorporating the CLBlast library into the framework. Different colors represent matrices of different size. Y-axis is the square matrix multiplication time in microsecond (us). Left section is the CLBlast program running as a normal OpenCL program. Right section is the performance of incorporating it into the Tensorflow framework.

To understand the overhead introduced by Tensorflow, the computational time was measured by incorporating it into the Tensorflow framework versus running it as a normal OpenCL program. The results are shown in Figure 13.

A.4. Problem encountered

Although a single run of Tensorflow MatMul operation was successful on mobile GPU, continuous runs of such operation resulted in unexpected situation. The program halted with no error message thrown and the process was killed by Android OS after several seconds. The reason of such unexpected error remained unknown and required further investigation.

B. Experiment —- OpenCL kernel optimization

In this section, several OpenCL kernel optimization techniques are tested. All experiments carried out here were based on the `*opencl-matmul*` program.

B.1. Base line performance

In this experiment, the simplest OpenCL MatMul kernel (called version 1) was tested against the CPU implementation. The performance is shown in Figure 14. Kernel version 1 is considered as the baseline performance because it's the most straightforward matrix multiplication kernel available. The programming methodology is as follow, a work-item is responsible for an element in the multiplied matrix. Each work-item performs row-column element-wise multiplication independently and sequentially just like normal human. The performance is roughly 2~3 times slower than CPU with matrix size equals to 1024.

B.2. Local memory

The usage of local memory gives better performance because the access latency is lower [1]. With this concept in mind, a new kernel called MatMul kernel version 2, was developed and used 16 by 16 2D local memory. The performance improvement between kernel version 1 and version 2 can be observed in Figure 15. The usage of local memory dramatically decreases the number of bytes loaded by a work-item. The performance is ~ 2 times faster than kernel version 1 given matrix size equals to 1024.

B.3. Transpose before Multiplication

Inspired by the matrix multiplication example in Qualcomm Adreno SDK, given a matrix multiplication task C=A*B, all matrices are stored in row-major arrays (default configuration in Tensorflow). The access pattern to matrix B isn't aligned. Such access pattern is considered bad because of low cache hit rate.

The engineering challenge is that no matter how we arrange both of the matrices (A and B), one of them must be accessed in an unaligned manner. The solution to such problem is to transpose matrix B before the matrix multiplication. As a result, the access pattern to B_T (transposed) matrix is aligned and the cache hit rate is high. This optimization technique comes at the cost of additional matrix transpose operation. From Snapdragon profiler, the L2 cache read hit rate of this transposed-before-multiply kernel reaches ~96% for a 64*64 square matrix multiplication task.

Also, due to the design limitation, this kernel is designed to be a 1D kernel, each work-item is mapped to a

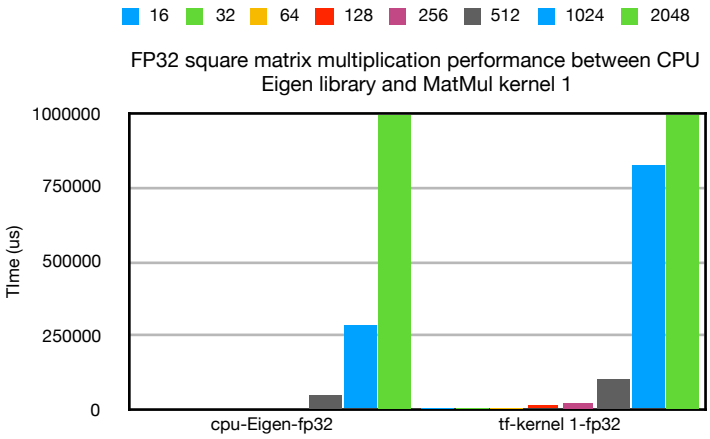

Figure 14. The FP32 square matrix multiplication performance between CPU Eigen library and OpenCL kernel version 1. Different colors show matrices of different size. Y-axis is the time needed in microsecond (us).

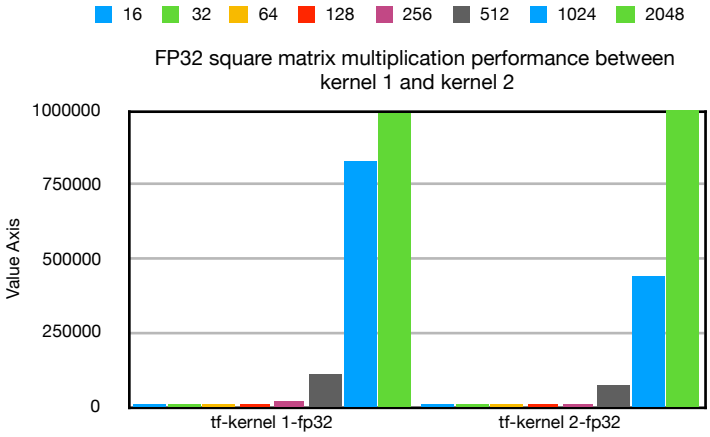

Figure 15. The FP32 square matrix multiplication performance between OpenCL kernel version 1 and OpenCL kernel version 2. Different colors show matrices of different size. Y-axis is the time needed in microsecond (us).

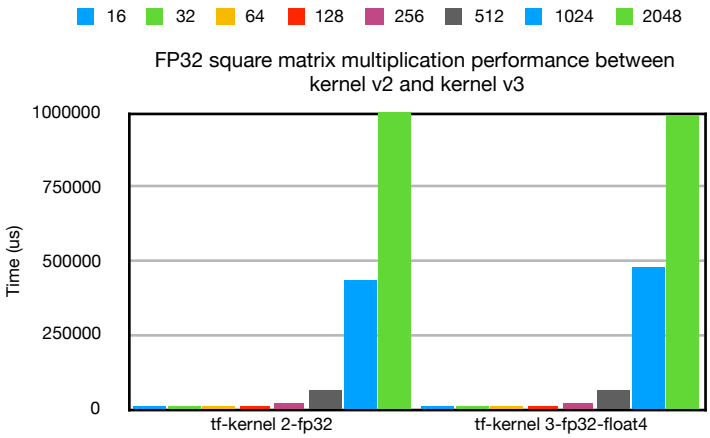

Figure 16. The FP32 square matrix multiplication performance between OpenCL kernel version 2 and OpenCL kernel version 3. Different colors show matrices of different size. Y-axis is the time needed in microsecond (us).

row in the multiplied matrix. Each work-item caches a piece of data into the local memory (coalesced memory access) and shared with all the work-items within the same work-group to minimize the memory load operation per work-item.

In addition, this kernel fully utilizes the memory bandwidth by vectorized load. The memory bandwidth of Adreno 540 system is 128 bits, which equals to float4 datatype. Thus, all memory load operation in this kernel

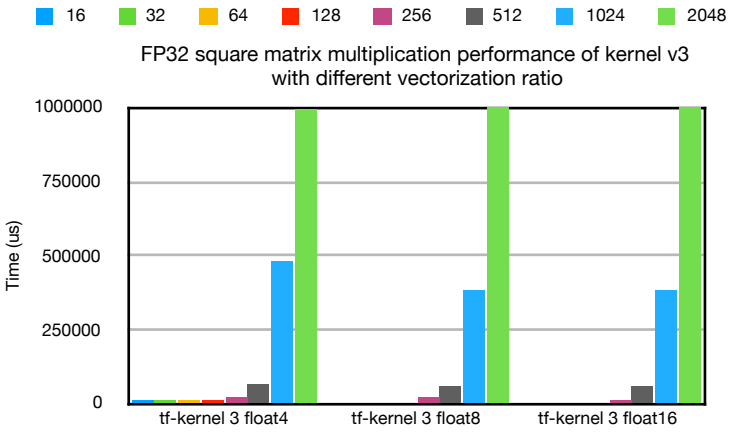

Figure 17. The FP32 square matrix multiplication performance of OpenCL kernel version 3 with different vectorization ratio. Different colors show matrices of different size. Y-axis is the time needed in microsecond (us).

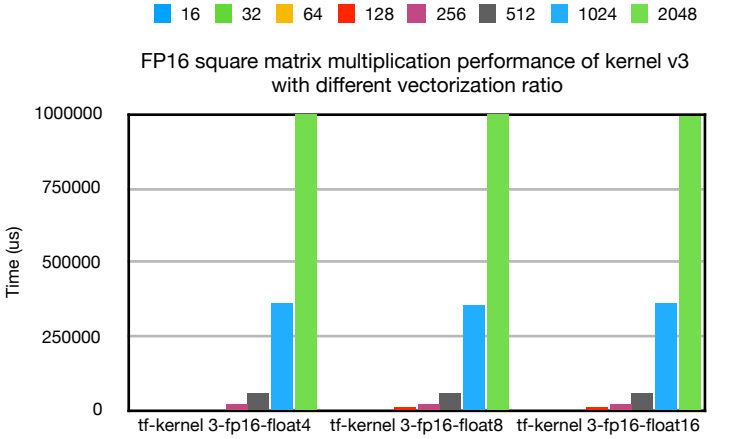

Figure 18. The FP16 square matrix multiplication performance of OpenCL kernel version 3 with different vectorization ratio. Different colors show matrices of different size. Y-axis is the time needed in microsecond (us).

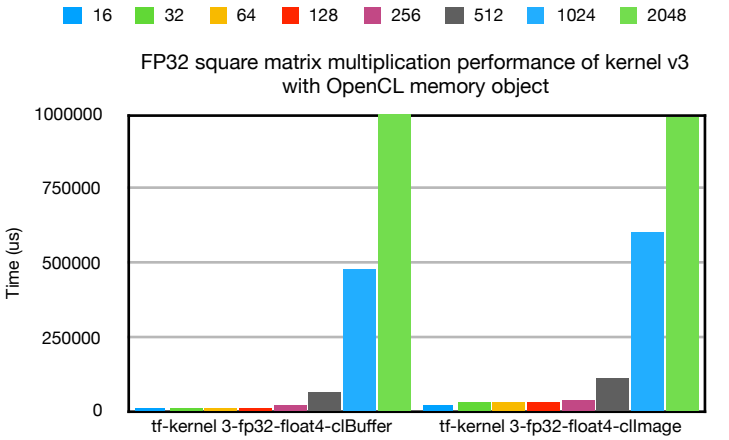

Figure 19. The FP32 square matrix multiplication performance of OpenCL kernel version 3 with different OpenCL memory object. Different colors show matrices of different size. Y-axis is the time needed in microsecond (us).

was designed to load 4 FP32 values from memory each time.

Each work-item writes to a single element in the multiplied matrix because the coalesced memory store to global memory isn't supported on Adreno 5xx series GPU.

With all optimization techniques mentioned above, the performance between kernel version 2 and the newly developed kernel 3 is shown in Figure 16. Out-of-expectation, the new kernel is worse than kernel 2.

B.4. Vectorization

The result from previous section gave no obvious improvement. The vectorization ratio was further increased to observed the differences. The vectorization ratio was increased from 4 to 16 to observer the best ratio. Results shown on Figure 17. The best vectorization ratio float datatype for this kernel is 16.

B.5. Workgroup size

Work group size is an OpenCL device dependent parameter. It should be tuned for a new device because the performance isn't portable. The work group size is related to the workload for each work-item. GPU stays idle most of the time given suboptimal work-group size, the amount of work distributed to GPU isn't able to keep it busy all the time. As a result, the advantage of latency hiding cannot be achieved and the performance is worse. On the contrary, given an over-estimated work group size, the performance remains the same because the maximum throughput has been reached. Further increase the work group size gives no better performance. On Table 1, optimal work group size for Adreno 540 GPU was found for different vectorization ratio of kernel 3.

Kernel name	*WG size*
tf-kernel 3-fp32-float4	16
tf-kernel 3-fp32-float8	16
tf-kernel 3-fp32-float16	64

Table 1. The optimal work group size for MatMul kernel 3 with different vectorization ratio.

B.6. Different OpenCL memory object

This optimization technique is specific to Adreno GPU because of its GPU architecture. The optimization trick was mentioned in a blog post on Qualcomm developer network [7]. In order to fully utilize the existing cache system, given a C=A*B matrix multiplication problem, matrix A is allocated as an OpenCL image object while matrix B is created as a normal OpenCL buffer. The methodology of such operation is to fully utilize the L1 cache located on the texture processor. Ideally, with the help of L1 cache, fewer memory traffic will pass to the system memory and the overall performance can be increased.

However, replacing the existing OpenCL buffer object with image object is troublesome because the carefully designed kernel is incompatible. The compromised option is to create an OpenCL image memory object from the existing buffer object. The results of such operations is shown on Figure 19.

The Figure shows performance reduction. After careful investigation into the GPU L1 and L2 cache hit rate, it's observed that it's impossible to create a true OpenCL image object from OpenCL buffer. The converted OpenCL image object is treated as a normal buffer object and nothing was loaded into the texture processor or L1 cache memory. Perhaps a new MatMul kernel should be developed to validate the possibility of such optimization technique.

B.7. FP16 over FP32

Claimed by Qualcomm, the throughput of FP16 is doubled compared to FP32. Additionally, data size is half of FP32, which further shorten the memory transfer time. The purpose of this experiment is to investigate the possibility of training the MNIST model on mobile GPU with FP16 precision.

A new class `*clQualcommFP16Engine*` was created with the following modification. All FP32 data would be converted to FP16 equivalent before the matrix multiplication. Matrices filled with FP16 values were passed to the MatMul kernel arguments. The FP16 kernel is similar to the FP32 one with slight modification on memory load/store operation. To save the effort of conversion, a FP32 variable was used to store the results of FP16 multiplication. Each element in the multiplied matrix if of type FP32. The result is shown in Figure 18.

On Figure 18, there's minor improvement in speed but the quality deteriorated as the number of matrices grew. For square matrix multiplication of size greater than 64. The per element accumulated error reached 0.1. Depending on the range of matrices data, the error fluctuated and the result wasn't stable. Furthermore, FP16 MatMul implementation cannot be applied to an AI training task because multiple sequential matrix multiplication results in unacceptable error. Despite the fact that this is the best performance achieved, the training task in the following section will be tested with a FP32 MatMul kernel.

B.8.Miscellaneous

Other optimization techniques have been implemented but no obvious performance improvement was observed. In this section, the miscellaneous optimization techniques are discussed including avoid the usage of `*size_t*` in kernel code, avoid integer module operation, use fast integer multiplication.

The reason why `*size_t*` data type should be avoided in an OpenCL kernel is the complexity of computing 64 bits integers. The `*size_t*` datatype will be promoted automatically by the compiler to 64 bits integer on 64-bit OS. Adreno GPU has to emulate a 64 bits integer with two 32 bits registers. The additional resource consumption is unnecessary if it can be replaced with other datatype.

All integer variables in the MatMul kernel was defined with the smallest functional datatype. The resource allocated for a variable just meets the required range of operation. For instance, it's impossible for matrices size to exceed 2^16=65535. Thus, all related variables were defined with the `cl_ushort` datatype, which to some extent, might reduce the computation and memory transfer time. Nonetheless, no obvious improvement was observed.

Integer module operation is expensive and another way to get the same result is binary AND operation. A mod 4 operation is equivalent to a binary AND operation with 3 (0x11).

Integer multiplication is expensive in Adreno GPU. If the expected result falls within the range of [-2^23, 2^23-1] (singed) or [0, 2^24-1] (unsigned), the `*mul24*` instruction is faster because fewer bits are calculated. However, the replacement of the `*mul24*` instruction gave minor performance improvement.

C. Training MNIST dataset with various AI models

With all MatMul OpenCL kernel tested in the previous section, training an AI model on mobile GPU is feasible and the result will be discussed in this section.

C.1.The design of pure training program

The purpose of this pure training program is to measure the time needed for training. Batches of training data will be loaded into Tensorflow runtime for computation. After the training process is done, the time passed will be calculated. Eventually, the testing samples will be loaded for 100 samples at a time. The overall accuracy is accumulated and averaged for the final model accuracy.

C.2.The design of training logger program

During the development process, it's hard to debug an AI training program without understanding the current

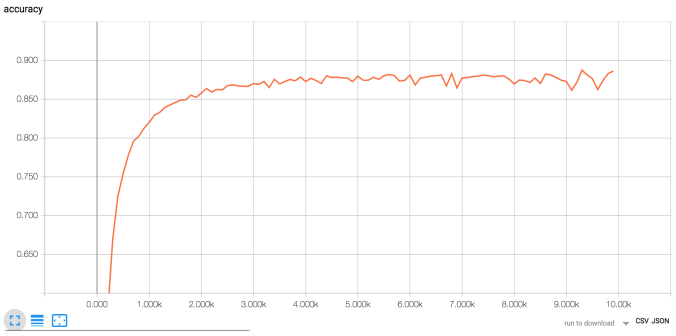

Figure 20. The MLP training accuracy on desktop computer.

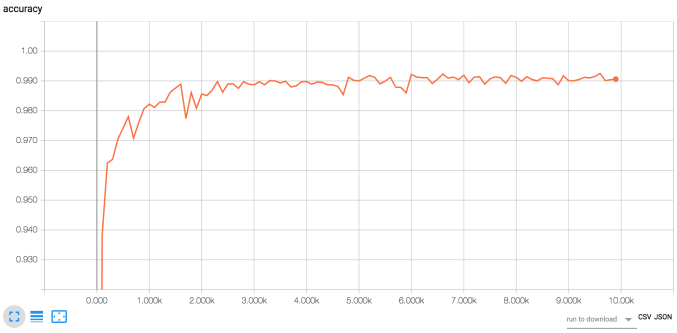

Figure 21. The DNN training accuracy on desktop computer.

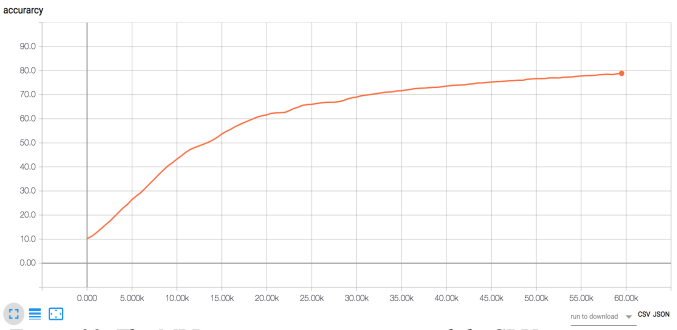

Figure 22. The MLP training accuracy on mobile CPU.

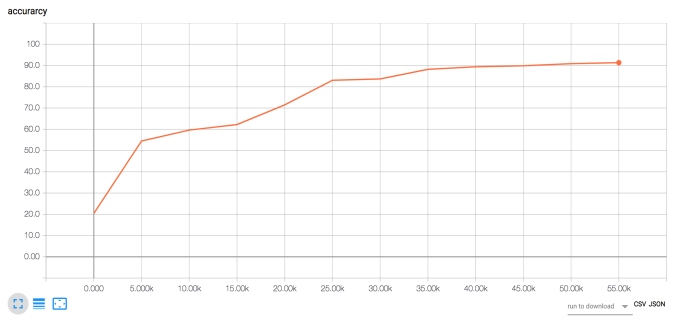

Figure 23. The DNN training accuracy on mobile CPU.

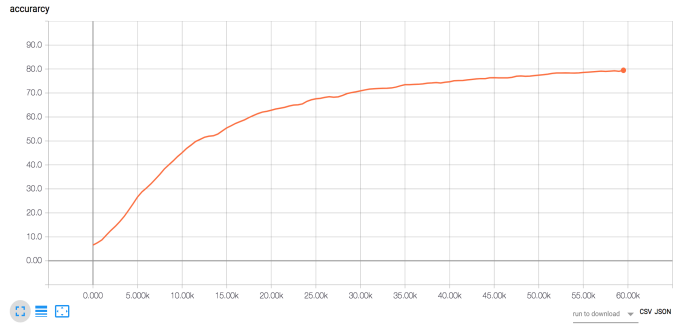

Figure 24. The MLP training accuracy on mobile GPU.

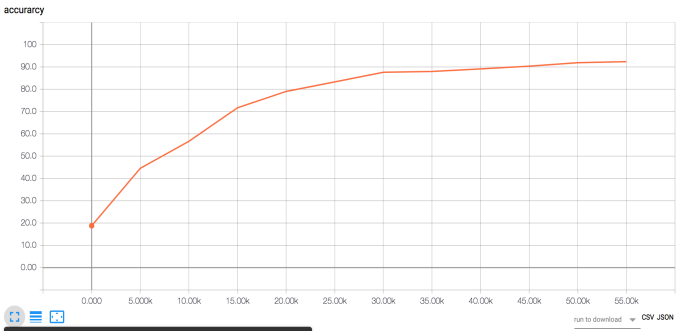

Figure 25. The DNN training accuracy on mobile GPU.

training accuracy. This program is designed to probe the trained model after a batch of training data is used to trained the model. The probed accuracy will be logged on mobile devices and viewed on desktop computer to inspect the training progress over iterations. Expectedly, this program is time consuming because testing a model for each training batch is computationally expensive. In the following discussion, the training progress on desktop computer is set as the ground truth for comparison.

C.3. Training accuracy

Figure 20 and Figure 21 show the MLP and DNN model training progress on desktop CPU. Figure 22 (MLP) and Figure 23 (DNN) show the training progress on mobile CPU. Figure 24 (MLP) and Figure 25 (DNN) show the training progress on mobile GPU. From Figure 20 to 28, it's obvious that the training results on mobile GPU is equivalent to the results obtained from desktop computer or mobile CPU. Which further proves that training on mobile GPU is successful.

C.4. Training time

The pure training performance on mobile CPU is shown in Table 3. The pure training performance on mobile GPU with different MatMul kernel implementations are shown from Table 4 to 6. Notice that the batch size is different for MLP and DNN model because the Tensorflow optimizer cannot reach a convergent result given large batch size in MLP model.

Observed from the results, the training accuracy are the same for both mobile CPU and GPU. The performance of CPU is still way faster than mobile GPU. The explanation for such phenomenon will be discussed in the discussion section.

Compared the results on Table 5 and 6, The DNN training time decreased by 26% because of higher vectorization ratio. At the same time, the MLP training

Table 3. Training performance on mobile CPU

Model Name	*Overall Accuracy (%)*	*Training Time (s)*	*Batch Size*
MLP	78.3435	5.34335	100
DNN	96.7990	216.708	1000

Table 4. Training performance on mobile GPU

Model Name	*Overall Accuracy (%)*	*Training Time (s)*	*Batch Size*
Kernel Used	*`MatMul_TN_1D_Fp32_Float4` + `MatTrans_1D_Fp32_Float4`*		
MLP	79.2828	56.3333	100
DNN	96.7909	508.305	1000

Table 5. Training performance on mobile GPU

Model Name	*Overall Accuracy (%)*	*Training Time (s)*	*Batch Size*
Kernel Used	*MatMul_TN_1D_Fp32_Float8` + `MatTrans_1D_Fp32_Float8*		
MLP	80.0404	56.5589	100
DNN	97.2151	527.089	1000

Table 6. Training performance on mobile GPU

Model Name	*Overall Accuracy (%)*	*Training Time (s)*	*Batch Size*
Kernel Used	*MatMul_TN_1D_Fp32_Float16` + `MatTrans_1D_Fp32_Float16*		
MLP	78.8788	53.1919	100
DNN	96.6364	388.855	1000

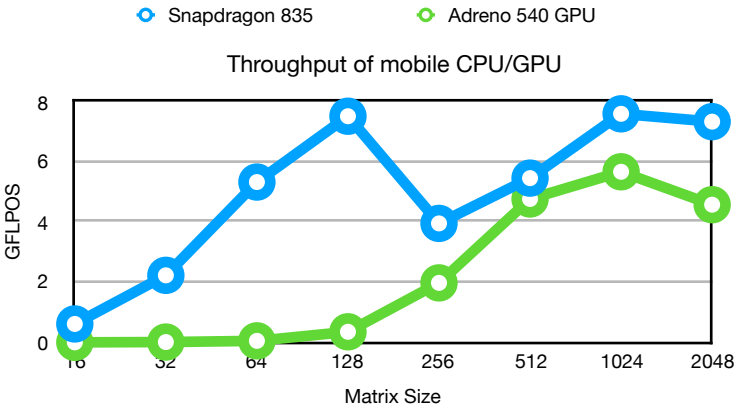

Figure 26. The throughput of Snapdragon 835 CPU and Adreno 540 GPU in square matrix multiplication task. X-axis is the size of the matrix. Y-axis is the throughput in GFLOPS.

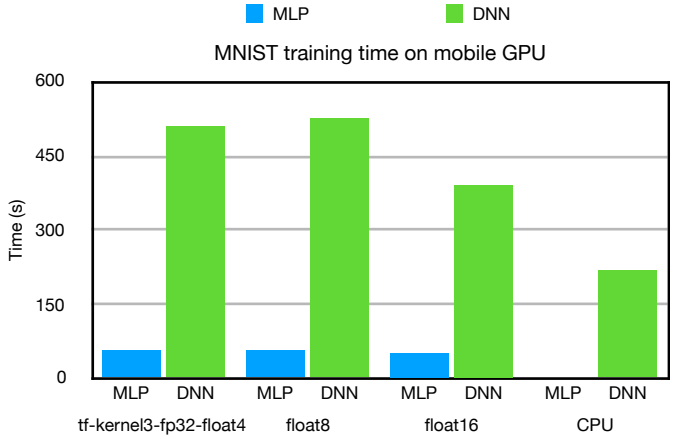

Figure 27. The MNIST training performance. Blue bar represents the time for MLP, and green for DNN. Y-axis is the time needed in second .

time decreased by merely 6%. Figure 27. visualizes the results of training time on mobile GPU.

D. GPU Computing capability:

This section explores the computing capability of mobile GPU/CPU by measuring the floating point operations per second in the square matrix multiplication benchmark.

From GeekBench [8], the throughput for Snapdragon 835 CPU is roughly about 11.5 GFLOPS. Based on our experiments, the throughput of was calculated as follow. For a square matrices multiplication task, the number of floating point operations roughly equals to N^3. As a result, the computed throughput is shown on Figure 26. The result obtained is slightly lower than GeekBench's measurement, the maximum throughput for CPU is ~7 GFLOPS, and GPU is ~ 6 GFLOPS.

VI. DISCUSSION OF RESULTS

A. Experiment — CLBlast evaluation

The tuned version of CLBlast OpenCL BLAS library is by far the fastest kernel tested on Adreno 540 GPU. In Figure 12. the performance of the tuned CLBlast library is slower than CPU if matrix size is smaller than 1024. The reason is as follow, profiled by the Snapdragon profiler, the actual computation consists of a small portion of time. A large portion of time (1272428 us ~= 1.2 sec) was spent on the compilation of OpenCL kernel source code. Since the performance measurements in Figure 12. were averaged for 10 iterations. As a result, given the actual computation consists of a small portion of time (i.e. matrix size < 1024), the kernel compilation time boosts up the average time significantly.

The compilation of BLAS OpenCL kernel source code in CLBlast library could be further identified by the measurement of Tensorflow overhead in Figure 13. The excessive time is contributed by the compilation of kernel source code. The CLBlast library is designed in a smart way such that the compilation process is needed only for the first run. The compiled binary will be cached in the system and a new OpenCL program will be created from binary instead of from source.

B. Experiment —- OpenCL kernel optimization

Among all kernels implemented in this paper, the 1D kernel with `transpose before multiply` method gives the best performance. For different ratio of vectorization, the

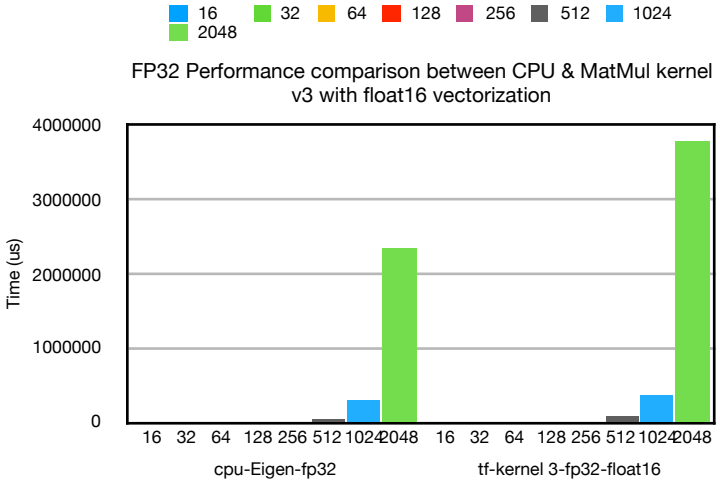

Figure 28. The FP32 square matrix multiplication performance of OpenCL kernel version 3 with float16 vectorization and CPU. Different colors show matrices of different size. Y-axis is the time needed in microsecond (us).

float16 data type is the most efficient. In additional, miscellaneous optimization techniques were applied. Notice that not all optimization strategy was integrated successfully, changing from FP32 multiplication FP16 wasn't successful because of the deteriorated precision, and replacing memory object from OpenCL buffer with image gave worse performance. Combined, MatMul kernel version 3 (FP32) with vectorization ratio of 16 gives the best performance among all manually designed kernels (CLBlast excluded). The performance comparison is shown on Figure 28. Still, mobile GPU is slower than mobile CPU. However, some additional factors should be taken into considerations including the theoretical throughput of mobile CPU/GPU, and the memory transfer time between host and OpenCL device.

C. Training MNIST dataset with various AI models

Compared with the MNIST training results on desktop computer, mobile CPU or GPU is capable of reaching the same model accuracy. The difference of growth rate between mobile platform and desktop platform is unexpected, the training accuracy increases dramatically on desktop computer while it grows slowly on mobile platform. Perhaps there're some API level optimization for Tensorflow Python API that causes the difference.

VII. LIMITATION

The limitations of this paper is separated into software part and hardware part.

For the software design of clMatMulEngine, an OpenCL context object is created for each matrix multiplication operation in Tensorflow runtime. Many OpenCL host side objects are created and released after computation. Such design choice isn't efficient because a context object, device object, command queue object can be reused in the next operation. In other words, the OpenCL host side objects should be kept for the same OpenCL devices. Host-side initialization should only be done once for a device. During the research phase of this project, such limitation was identified. The original initiative was to build a well-integrated OpenCL version of Tensorflow. The plan was cancelled because the estimated amount of engineering work is beyond the workload of this project. Such operation requires deep integration of OpenCL into the Tensorflow framework.

For the hardware limitation, the closed source architecture of Adreno GPU makes it challenging to verify the optimization strategy. For instance, the info about the size of the on-chip local memory, the size of L2 cache, the size of L1 cache aren't revealed by Qualcomm.

VIII. ACKNOWLEDGEMENT

We would like to give special thanks to Prof. C.L. Wang, Department of Computer Science, the University of Hong Kong for his guidance and help. The former intern 张启萌 who shared his work on the Caffe library on Snapdragon 820 platform. PhD student Pengfei Xu who shared his experience on deep learning benchmark on GPU. Final year student Ji Zhuoran who shared his experience on porting Caffe library to mobile GPU. Student Liu Weizhi who shared his work on GPU kernel stretching and slicing on NVIDIA GPU. We would like to give thanks with whole heart to anyone who contribute and share idea with us.

IX. CONCLUSION

The advancement of computing power on mobile devices and the recent progress in AI push the computation toward the users' end. In this report, the possibility of training AI models on mobile devices was explored by embedding OpenCL code into the Tensorflow framework. Also multiple benchmarks were tested on the mobile platform to understand the characteristics of the heterogeneous computing platform. The training and inference processes on mobile devices were accelerated by off-loading the intensive computation from mobile CPU to GPU. Also, training a MNIST dataset on mobile GPU was successful. Despite the matrix multiplication task was slower on mobile GPU. The best version of the manually designed OpenCL kernel outperformed the baseline performance by 2.16 times for square matrix multiplication of size 1024. Further investigation is needed to unveil the underlying hardware architecture of mobile GPU, and explore the capability of mobile AI applications.

X. REFERENCE

1. Qualcomm Snapdragon(TM) Mobile Platform OpenCL General Programming and Optimization. 2017.
2. Khronos. *The OpenCL Specification*. 2018; Available from: https://www.khronos.org/registry/OpenCL/specs/opencl-2.2.html.
3. Moustafa, A., et al., *RSTensorFlow: GPU Enabled TensorFlow for Deep Learning on Commodity Android Devices*. Proceedings of the 1st International Workshop on Deep Learning for Mobile Systems and Applications. 2017: ACM.
4. McVoy, L.W. and C. Staelin. lmbench: Portable Tools for Performance Analysis. in USENIX annual technical conference. 1996. San Diego, CA, USA.
5. Konstantinidis, E. and Y. Cotronis, A quantitative roofline model for GPU kernel performance estimation using micro-benchmarks and hardware metric profiling. Journal of Parallel and Distributed Computing, 2017(107): p. 37-56.
6. Nugteren, C., *CLBLast: A tuned openCL BLAS library.* arXiv preprint arXiv:1705.05249, 2017.
7. Qualcomm. *Matrix Multiply on Adreno GPUs*. 2016; Available from: https://developer.qualcomm.com/blog/matrix-multiply-adreno-gpus-part-1-opencl-optimization.
8. GeekBench. *The Qualcomm Snapdragon 835 Performance Preview*. Available from: https://www.anandtech.com/show/11201/qualcomm-snapdragon-835-performance-preview/2.