香　港　大　學

THE UNIVERSITY OF HONG KONG

Course Info: ELEC4848 Senior Design Project 2017-2018

Project Title: The performance optimization on TensorFlow framework on Mobile GPU devices using OpenCL

Supervisor: Prof. C. L. Wang (clwang@cs.hku.hk)

Name: Cheng Wei (wei_cheng@hku.hk)

Date of Submission: 2018.04.09

Summary

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 report, 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.

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.

Table of content

List of Figures

List of Tables

Abbreviations

- TF: Tensorflow
- GPU: Graphical Processing Unit
- CPU: Central Processing Unit
- GEMM: GEneral Matrix-to-matrix Multiplication
- AI: Artificial Intelligence
- MLP: Multi-Layer Perceptron
- DNN: Deep Neural Network
- MNIST: Modified National Institute of Standards and Technology
- OpenCL: Open Computing Language
- BLAS: Basic Linear Algebra Subprogram
- FP32: Single Precision Floating Point
- FP16: Half Precision Floating Point
- TDP: Thermal Design Power
- SPMD: Single Program Multiple Data
- SoC: System On Chip
- PIE: Position Independent Execution

I. INTRODUCTION

The introduction section covers the project motivation, project background, project goal, report organization, project deliverables, project schedule, and literature review.

A. Motivation

As researchers push the boundary of AI (Artificial Intelligence) and the blossom of mobile devices, it's foreseeable that the usage of AI might soon swing toward the users' end for better user experience. For instance, Apple's faceID technology stores biological data locally on the devices and performs computationally intensive AI training task locally to adapt to users' facial changes. If a piece of facial biological data is captured on your iPhone and sent back to Apple server for AI analysis, phone users might be concerned about the privacy or security issue. If an AI model is trained locally on users' devices, privacy will no longer be a concern. Therefore, in this report, the possibility of training and inference Tensorflow AI models on mobile devices with the help of mobile GPU (Graphical Processing Unit) will be explored.

B. Background

Tensorflow is popular among both industry and researchers 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.

Conventionally, an AI model training process is more computationally intensive than the inference process. To accelerate both operations, it's desirable to leverage the power of GPU by replacing the existing code with parallel computing language, OpenCL (Open Computing Language), which is an industry-backed, open-source framework for parallel computing across heterogeneous platforms such as CPU (Central Processing Unit), GPU, DSP etc. By training and analyzing a relatively simple but meaningful machine learning model on an Android device using mobile GPU, the possibility of edge computing in AI applications is demonstrated.

C. Project goal

The goal of this project is to train the MNIST (Modified National Institute of Standards and Technology) dataset with Tensorflow framework on mobile GPU using OpenCL.

D. Report organization

The report is organized in the following manner. The introduction section gives a brief overview of the overall structure. The second section, analyze of problem, identifies possible issues from both software and hardware perspectives. Next, the theoretical principles section introduces the GPU architecture and some commonly used optimization techniques used in GPU programming. After that, method of investigation section is added to elaborate how the problem will be analyzed by running various of benchmark tests. Then, the design and construction of software system gives details about the design of the abstract `*clMatMulEngine*` class and the optimization techniques applied to it. Results of existing or manually-designed testing programs are shown in the experiment results section. Followed by discussion section where results obtained are compared with the theoretical principles. Last but not least, the conclusion for this report will be listed.

E. Project deliverables

The deliverable of this project will a codebase capable of training AI models on mobile GPU. The training and inference processes should be accelerated by the mobile GPU using OpenCL.

F. Project schedule

In the first semester, a considerable amount of time has been spent on understanding the software architecture of Tensorflow, choosing suitable hardware platforms for experiments, and collecting information. In the second semester, intensive coding and implementation effort have been made to collect experimental data.

G. Literature review

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

RSTensorflow [1] 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. RenderScript is a programming framework designed by Google to support parallel computation on Android devices. In order to support a wide range of Android devices, the underlying hardware architecture is hidden from the programmers. On the other hand, OpenCL only supports Android devices with OpenCL driver; however, it has more control over the underlying hardware, which means better performance can be expected. Instead of RenderScript, OpenCL is chosen in this project because of its efficiency.

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 [2] 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.

II. ANALYSIS OF PROBLEM

A. Software — Tensorflow

A.1. Architecture

The Tensorflow software architecture is best illustrated in Figure 1 [3]. The top layer libraries such as training libraries, inference libraries, Python clients library, and C++ clients library all depend on Tensorflow C API. Behind it, the distributed master is designed to distribute the load across multiple workers, and the data flow executor is designed for the execution of computational graph. The kernel implementation libraries define all necessary mathematical operations needed in the Tensorflow library. In this project, the matrix multiplication operation is chosen as the optimization target because of its common usage in AI applications [4].

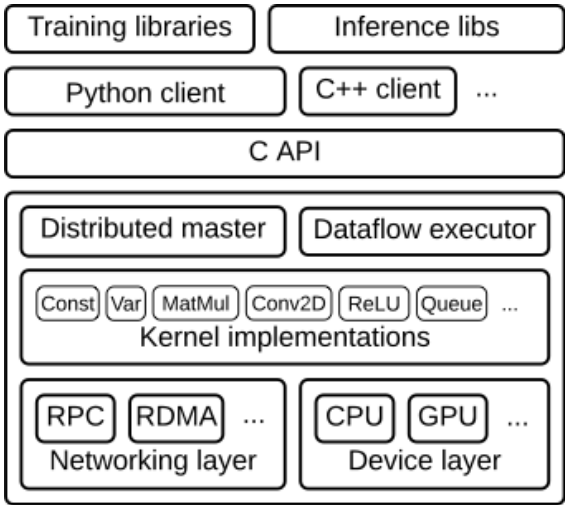

Figure 1. Tensorflow software architecture.

The core computation happens in the kernel implementations part, and most them depend on Eigen library, which is a C++ template library generating efficient code for GPU/CPU.

A.2.OpenCL in Tensorflow

Tensorflow doesn't support OpenCL directly, instead, it depends on the SYCL library, which is a C++ abstraction layer of OpenCL. Until the time of writing, there're three SYCL implementations available on the market. Namely, Computecpp [5] by Codeplay, triSYCL [6] leaded by Xilinx, and sycl-gtx by [7] [8]. Among all these implementations, only the sycl-gtx library is open-source and functional. In the first semester of 2017-2018, I found it nearly impossible to come up with a fully operational OpenCL implementation of SYCL. Therefore, instead of porting the backend of Tensorflow library, which is the Eigen library, to mobile GPU, OpenCL code was injected directly into the Tensorflow codebase.

A.3.Tensorflow on mobile platform

Google has been focusing on mobile AI application and aggressively incorporating the Tensorflow framework into their Android system. Until the time of writing, TensorflowLite [2] is the official support for AI inference task on mobile CPU, and Qualcomm supported Tensorflow AI model inference on its DSP and GPU [9, 10].

To achieve the project goal, which is training Tensorflow AI model on mobile GPU, several technical difficulties were identified. First, the code for AI training and inference is separated in the framework, deep understanding of the overall framework structure is necessary before making any contribution to it. Not to mention the time spent on digesting the industry-level codebase contributed by experienced developers around the world. Secondly, cross compilation is needed for any application running on mobile devices; however, the training code in the framework isn't designed for such purpose. The training code should be modified to run on mobile devices. Thirdly, the framework only supports desktop Nvidia GPU via cuDNN library and AMD GPU via SYCL library. Two main stream mobile GPUs on the market, ARM Mali GPU and Qualcomm Adreno GPU, don't support any of those, which means porting the codebase directly is impossible.

B. Hardware — Mobile GPU

The rise of GPU in AI application was driven by the trend of deep learning. 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.

Similar to the trend in desktop GPU market, performance of mobile GPU dramatically increases as mobile gaming gains momentum. Yet, there're still differences between the two. 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 [11] while desktop GPUs come with a piece of dedicated memory on chip separated from CPU.

There are two mainstream mobile GPUs in the market that support OpenCL: ARM Mali and Qualcomm Adreno. Until the date of writing, the latest ARM Mali GPU supports supports OpenCL 1.2 with full profile functionality [12], and the latest Qualcomm Adreno GPU supports OpenCL 2.0 with full profile functionality [11]. In this project, Qualcomm Adreno 540/530 GPUs were used because the company has better vendor support and better GPU profiling tools.

In general, in order to run a simple OpenCL program on mobile GPU, we need the following items. A cross compilation toolchain (in our case the Android NDK toolchain) to cross compile the binary code for Android device, and a list of OpenCL

headers (available on Khronos website), and a vendor specific OpenCL driver shipped with the phone [11].

Qualcomm Snapdragon profiler [13] is the official desktop profiling tool for CPU/GPU/DSP on Qualcomm chips. I can analyze the overall system level performance by observing the GPU load, GPU L1 cache miss rate, GPU L2 cache miss rate, OpenCL kernel enqueue time, kernel execution time, memory copy time, kernel instructions, distribution of kernel assembly code (e.g. percentage of NOP instruction in the OpenCL kernel code) etc.

C. Adreno GPU

Adreno GPU is chosen in this project, thus, more specific details will be covered in this section. Due to the fact that the technical implementation is proprietary, the architecture detail included in this section is merely a concise version of the content in the programming guide [11].

The illustrative architecture is shown on Figure 2. There're a shade processor (SP) and a texture processor (TP) in the system. Memory traffic generated by either processor has to go through L2 cache to the system memory. However, an OpenCL image object and a buffer object are treated differently in such design. Additional L1 cache is located in the texture processor and it's only accessible by an OpenCL image read operation.

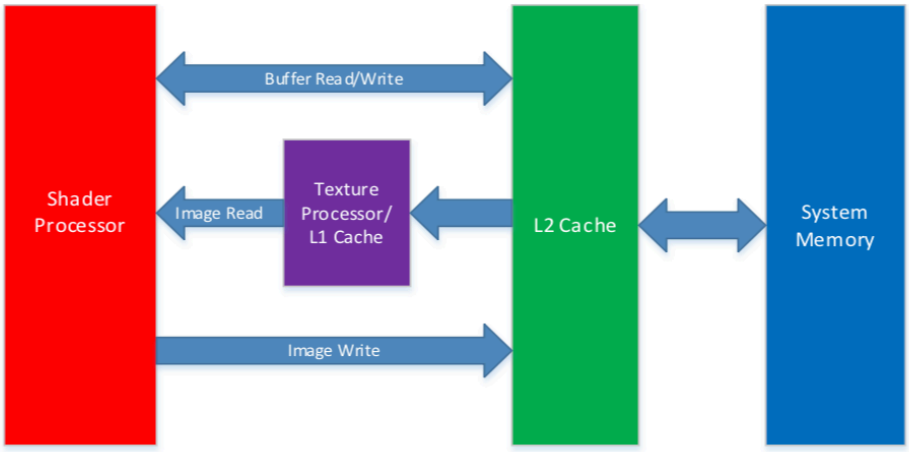

Figure 2. High-Level Adreno GPU architecture.

D. OpenCL

OpenCL is the industry driven SPMD (Single Program Multiple Data) programming model for GPU [14]. Unlike other mobile GPU programming framework such as RenderScript used in RSTensorflow [1], OpenCL exposed the underlying hardware to the developers which is more flexible and extensible.

The OpenCL framework is separated into following parts: platform model, execution model, memory model, and programming model. The platform model defines the high-level heterogeneous system. The execution model abstracts how SPMD commands are executed on the platform. The memory model defines different levels of memory in the heterogeneous system. The programming model defines a high level idea when designing an algorithm.

For the platform model, a host device is connected to one or more OpenCL devices. For instance, on mobile platform, the host is the CPU, the OpenCL device is the mobile GPU. An OpenCL device is divided into smaller parts called compute units (CUs), which are further divided into processing elements (PEs). The reason of defining such concept is for the easiness of explanation in the OpenCL execution model.

In the execution model, an OpenCL application can be separated into two parts, namely host-side code and kernel code. The host-side code is run on host device and the kernel code is executed on OpenCL devices. A host can submit a kernel for execution on OpenCL devices. At the same time an integer index space is created for kernel execution. An instance of kernel is defined as a work-item and identified by its global id. Furthermore, work-items are grouped into a work-group which is an abstract concept of bundled execution.

In the memory model, OpenCL defines two types of data storage: buffer objects and image objects. Buffer object is a block of adjacent memory just like an array. Image object is more complicated because of various image formats supported, and the implementation is vendor specific. Normally, a buffer object is easier to use but the kernel developer has to be very careful with the boundary cases. As for image object, it's easier to conceptualize when dealing with image processing kernel because it's 2D naturally. For the memory region, OpenCL defines five different memory regions, namely global memory, constant memory, local memory, private memory. In detail, global memory is accessible for all work-items in a work-group. Constant memory is a piece of memory defined as constant in global memory. Local memory is accessible for all work-items in a workgroup. Notice that the implementation of local memory is vendor specific, a local memory can be added next to an

OpenCL device or mapped to a section of global memory. Private memory is private to each work-item and invisible to others.

The programming model abstracts the high level algorithmic design concept. For instance, a developer can come up with a data-parallel algorithm given the problem itself is parallizable. On the other hand, a task-parallel programming model is achieved when kernels are submitted for an out-of-order queue for execution. The dependencies between the kernels are resolved by the runtime scheduling. The limitation of such programming model is the nature of the problem. For instance, given a naturally sequential problem, it's impossible to make it task-parallel.

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 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 patterns 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 the later part of this report. 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 3 and Figure 4 respectively.

From Figure 3 and Figure 4, 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 [16] 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

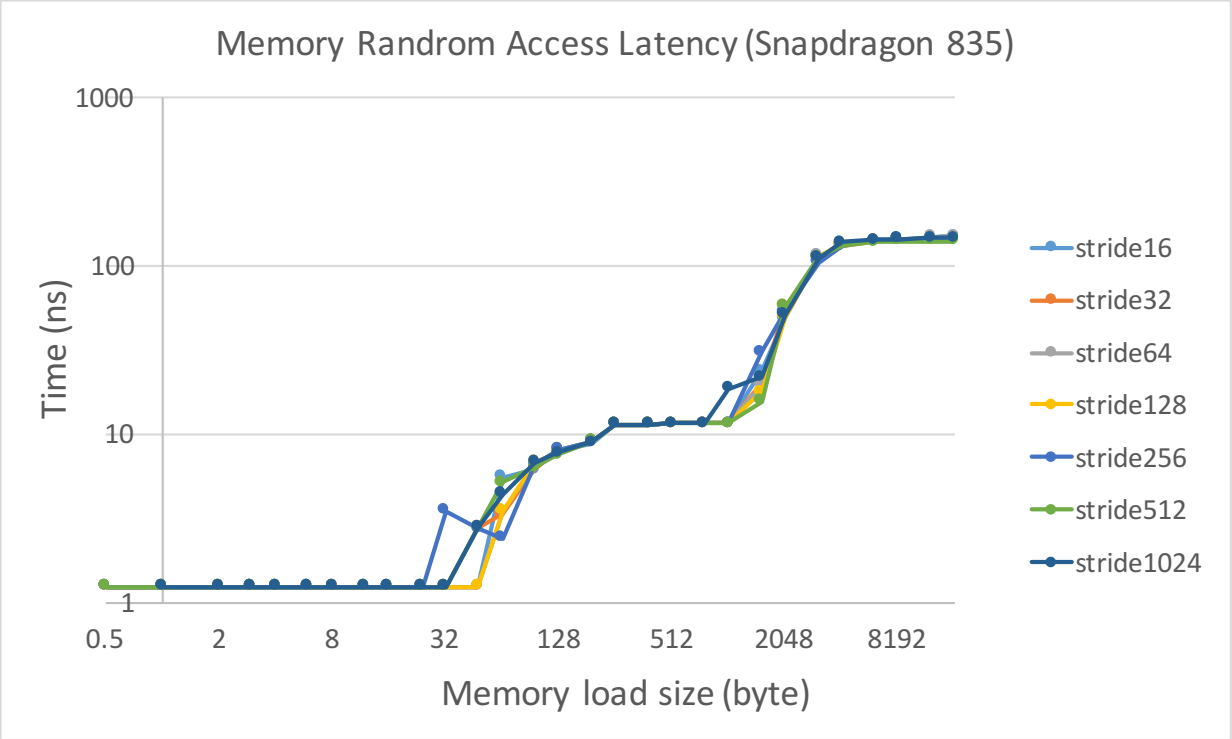

Figure 3. is the memory random access latency for Snapdragon 835 platform. X-axis is the memory load size in byte. Y-axis is the memory random access time in microsecond (us). Multiple data points on the right show the stride size for random memory access. Stride sizes from 16 bytes to 1024 bytes were performed in this experiment.

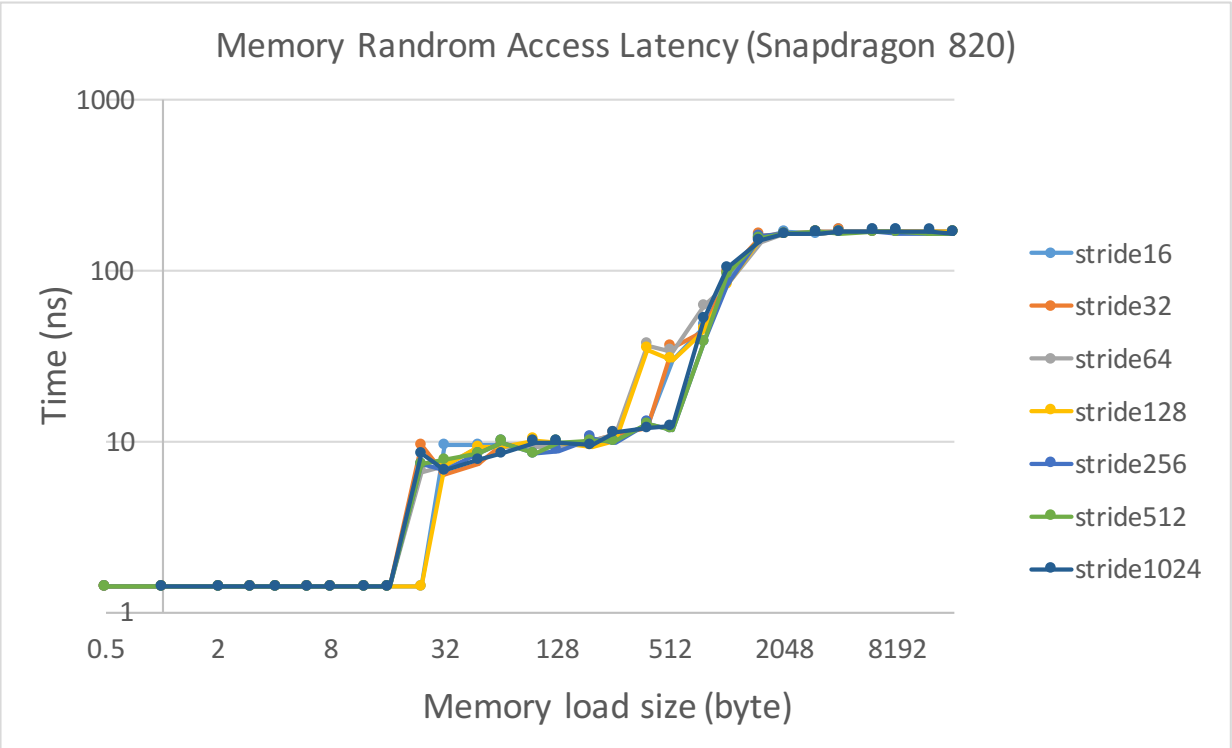

Figure 4. is the memory random access latency for Snapdragon 820 platform. X-axis is the memory load size in byte. Y-axis is the memory random access time in microsecond (us). Multiple data points on the right show the stride size for random memory access. Stride sizes from 16 bytes to 1024 bytes were performed in this experiment.

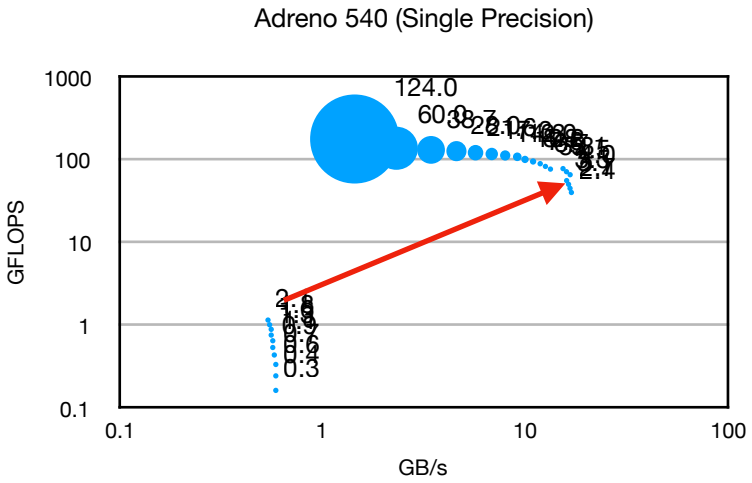

Figure 5. The performance of single precision floating point operation on Adreno 540 GPU. X-axis is the memory bandwidth in GB/s. The Y-axis is the throughput (GFLOPS). The diameter of the data point is the kernel operation intensity in FLOP/Byte. The number right next to the data point is the operation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

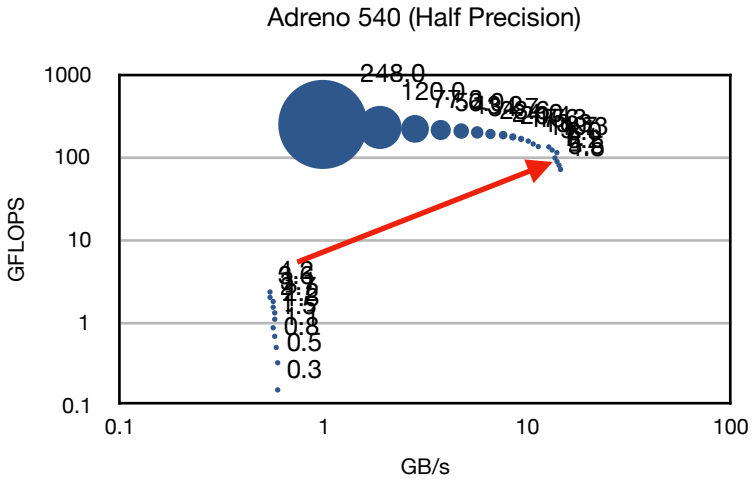

Figure 6. The performance of half precision floating point operation on Adreno 540 GPU. X-axis is the memory bandwidth in GB/s. The Y-axis is the throughput (GFLOPS). The diameter of the data point is the kernel operation intensity in FLOP/Byte. The number right next to the data point is the operation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

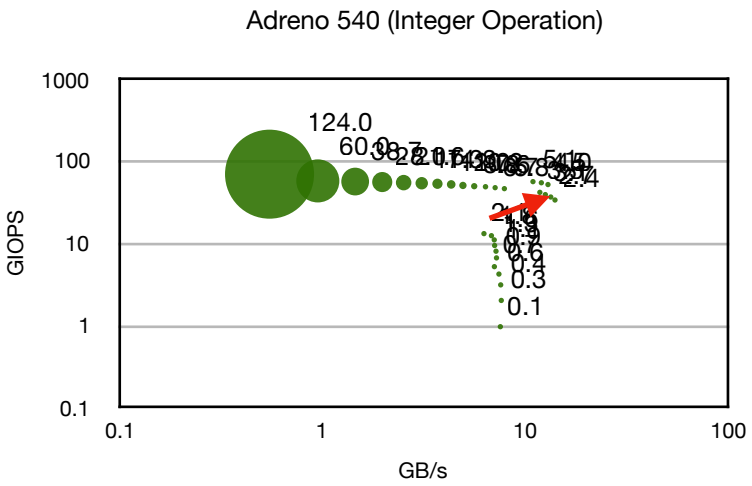

Figure 7. The performance of integer operation on Adreno 540 GPU. X-axis is the memory bandwidth in GB/s. The Y-axis is the throughput (GIOPS). The diameter of the data point is the kernel operation intensity in IOP/Byte. The number right next to the data point is the operation intensity.

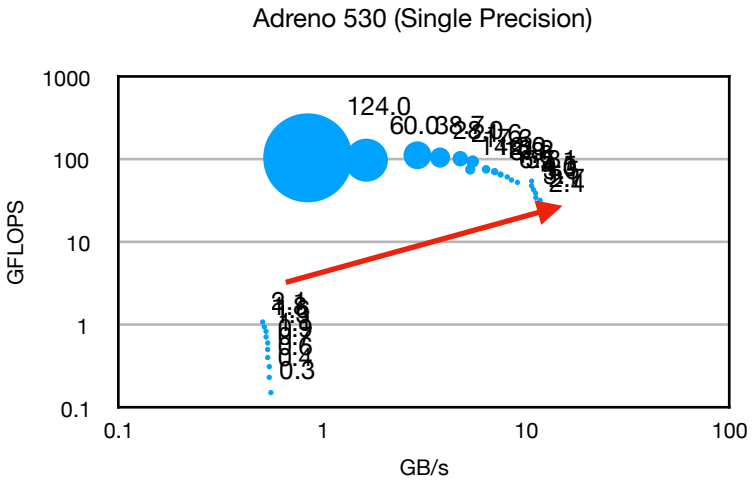

Figure 8. The performance of single precision floating point operation on Adreno 530 GPU.

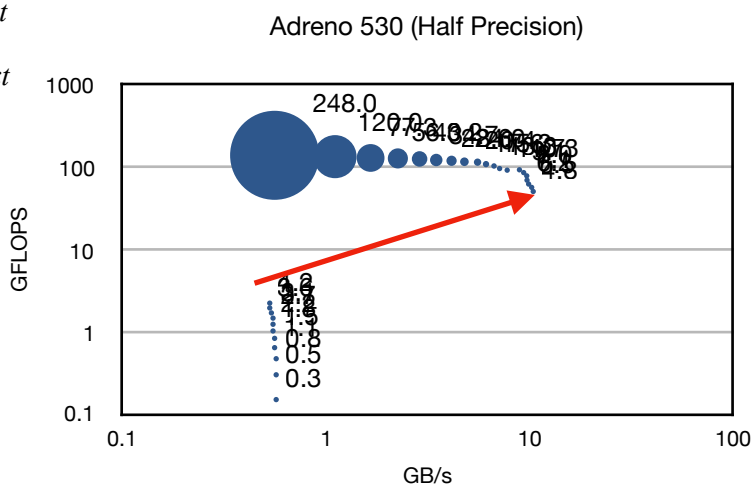

Figure 9. The performance of half precision floating point operation on Adreno 530 GPU.

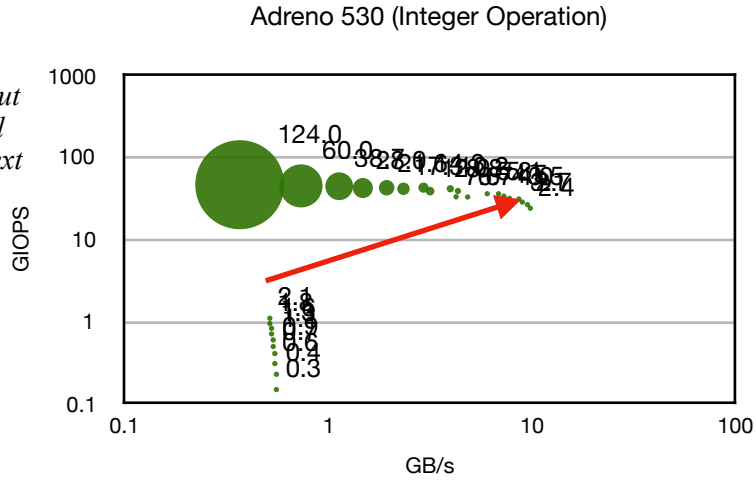

Figure 10. The performance of integer point operation on Adreno 530 GPU.

loaded into the GPU memory by the *`clEnqueueWriteBuffer`* function in the OpenCL 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

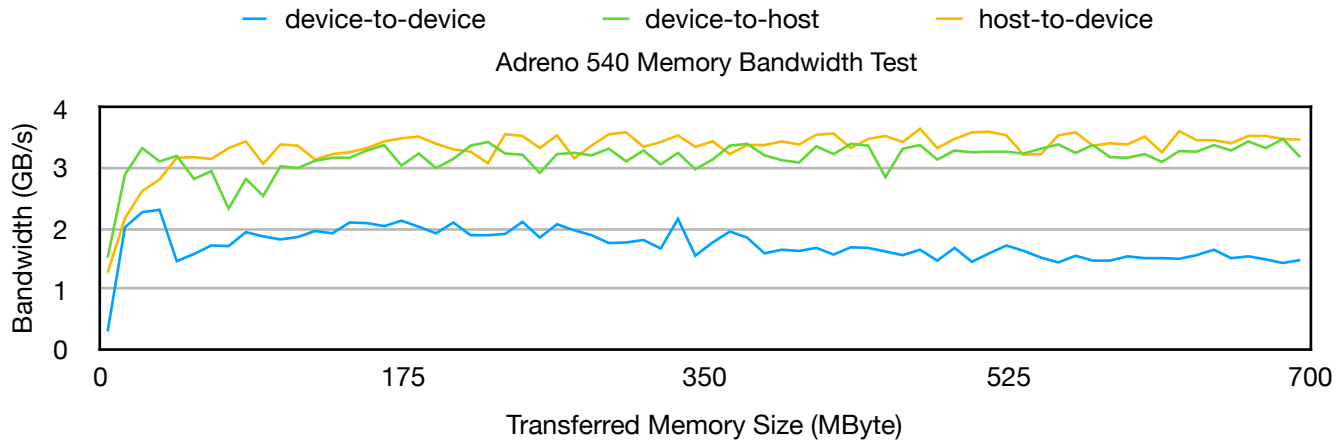

Figure 11. This chart shows the memory transfer bandwidth between host device and OpenCL device on Adreno 540 GPU. X-axis is the memory size being transferred in Mbyte. Y-axis is the measured bandwidth in GB/s.

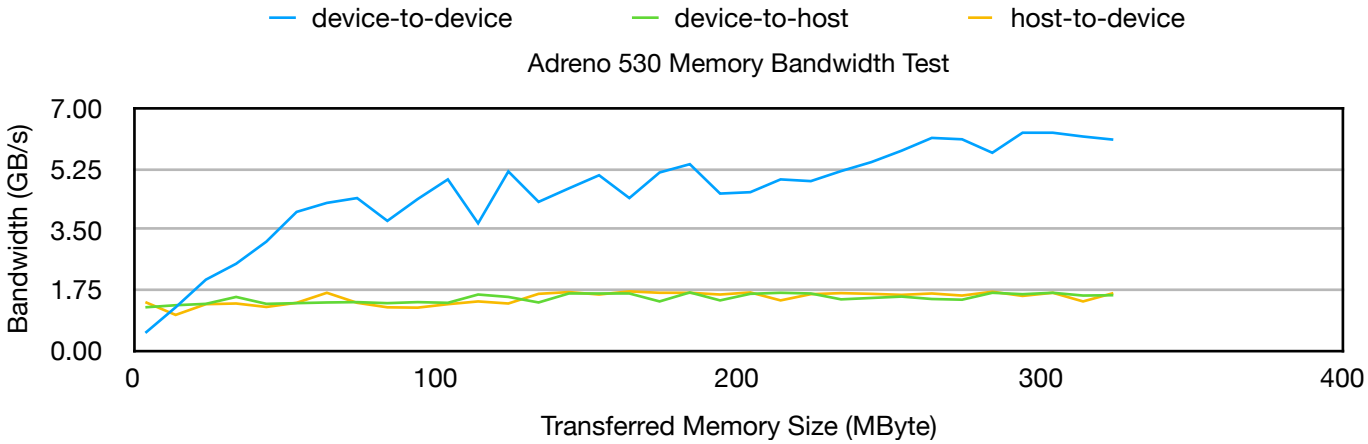

Figure 12. This chart shows the memory transfer bandwidth between host device and OpenCL device on Adreno 530 GPU. X-axis is the memory size being transferred in Mbyte. Y-axis is the measured bandwidth in GB/s.

scenarios. Data transfer from 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 11, and results of Adreno 530 GPU are shown in Figure 12. Notice the difference between Figure 11 and Figure 12, both Adreno 540 and Adreno 530 devices have similar host-to-device and device-to-host 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. The source code can be found in *Appendix G: OpenCL memory bandwidth test source code.*

V. PREPARATION WORK

Mentioned in the analyze of problem section, many technical barriers need to be overcome before any optimization techniques are applied. In this section, all preparation work will be covered.

A. Tensorflow porting effort

The most challenging part of porting Tensorflow training functionality to Android phone is to identify the problematic segments and modify the codebase with minimum code injection. Unlike conventional machine learning framework (i.e. Caffe) that depends on traditional GNU Make system, the Tensorflow framework relies on the Bazel software building tool developed by Google. In such system, a library or a binary are regulated by a BUILD file which is similar to the Makefile in GNU Make.

One of the several technical difficulties encountered during the development is the compilation of `*//tensorflow/core:android_tensorflow_lib*` target in the framework. The target itself contains most of the libraries needed for training an AI model. Several C++ versions have been tested to cross compile the existing codebase. Limited C++ support from Android NDK toolchain adds complexity to the development. For instance, the `*std::to_string()*` function hasn't been added to the NDK toolchain revision 15c, a temporary C++ template patch was

created for this specific function to pass the compiling process.

Another technical difficulty encountered is the denial of execution on Android phone. For Android version greater than 5.0, PIE (Position Independent Execution) is enforced to ensure the security. Program compiled without `*-fPIE -pie*` linker flags won't be allowed to executed on Android version newer than 5.0.

`*-lpthread*` is commonly used compiler flag for UNIX-like platform if the code shall be linked with the pthread library. Out of expectation, it is included in the Android NDK toolchain automatically and adding such flag will cause a linking error. Manual removal was carefully performed to ensure the configuration is correct for both the cross-compiled Android platform and the original desktop platform.

LMDB (Lightning Memory-Mapped Database) is a library that Tensorflow training codebase depends on. The cross compilation of such library requires an additional compiler flag `*-DANDROID*` and the removal of `*-lpthread*` linker flag.

B. Dataset preparation effort

MNIST handwritten digits dataset consists of 60000 training samples and 10000 testing samples. Each sample is a grey scale image of size 28*28. With wichtounet's [17] kind contribution to the open-source community, the work of parsing MNIST dataset and incorporating it into the mobile GPU training program is reduced dramatically.

C. Tensorflow AI model preparation effort

Tensorflow AI model is built by its Python API to reduce the programming complexity. A Tensorflow model file is a machine-readable protocol buffer file (.pb) which defines the computational graph of certain AI algorithm. The file was sent to Android devices and read by Tensorflow runtime to rebuild the computational graph structure on mobile devices.

In this project, two AI models will be presented, MLP (Multi-Layer Perceptron) and DNN (Deep Neural Network) respectively. The architectural design and hyper-parameters of such AI model is out of scope of this project, only the content will be covered. The first model, MLP (), consists of 1 input layer of size [batch_size, 784], 1 output layer of size [batch size, 10], 2 hidden layers, each with 50 neurons. The second model, DNN, consists of 2 convolutional layers (32 5x5 filers for the first and 64 5x5 filters for the second), 2 pooling layers (2x2 filter with stride 2 for both), 2 fully connected layers, 1 drop-out layer.

Since the system bottleneck and optimization target of this project are matrix multiplication, only the number of such operation is analyzed. For both DNN and MLP, the number of matrix multiplication operation is shown in Table 1. The Python script used to generate the models can be found in *Appendix B: MNIST AI model building Python script.*

Table 1: MatMul operation analysis

Model Name	MatMul Operation
MLP	[batchSize, 784] * [784, 50] * [50, 50] * [50, 10]
DNN	[batchSize, 3136] * [3136, 1024] * [1024, 10]

D. Tensorboard logger porting effort

Tensorboard is a Tensorflow application debugging tool developed by Google. It's only available on desktop platform using Tensorflow Python API. The debugging tool is essential for AI training processes and worth porting to Android platform. Inspired by tensorboard_logger project on GitHub [18], the MINST training program is capable of generating Tensorboard compatible log files with slight modification. Consequently, the training accuracy of an AI model on mobile devices was visualized using the powerful Tensorboard debugging tool.

E. Benchmark porting effort

For the thorough understand of the SoC performance, benchmark programs for both memory system and GPU were executed. For memory system benchmark, as suggested by my supervisor, the classical LMBench [15] was ported to Android platform. For GPU benchmark, the MixBench [16] was ported to mobile GPU.

The LMBench was originally designed for a UNIX-like system, and luckily, most of the tests could be ported to Android platform except some network related benchmarks. MixBench is a GPU benchmark tool testing mixed operational intensity kernels. Porting it was less troublesome because of well-supported cross compilation setup.

F. OpenCL kernel compiler

Offline kernel compilation is one of the OpenCL optimization technique applied in the later section. There're two ways to create an OpenCL program object: online compilation and offline compilation. Online compilation creates an OpenCL program object by calling `*clCreateProgramWithSource*`, and the offline compilation creates an object by calling `*clCreateProgramWithBinary*`. To support offline compilation, this OpenCL kernel compiler is designed from scratch. First, the compiler creates an object by reading the OpenCL kernel source code in bytes. The created OpenCL program object is then queried by the `*clGetProgramInfo*` function with `*CL_PROGRAM_BINARY_SIZES*` flag. The right amount of memory will be allocated based on the value returned. Next, read the program binary size using the `*CL_PROGRAM_BINARIES*` flag, and eventually, write it to a file in binary format. The source code can be found in *Appendix E: OpenCL compiler source code.*

G. Equipment preparation

In this project, the benchmark tests were conducted on multiple devices including Xiaomi 6 Android phone, Open-Q Snapdragon 820 development kit. Some trivial equipment preparation work is needed, including flashing the Android ROM, getting root access on Android platform etc.

Also, suggested by the Qualcomm programming guide [11], the power limitation on both mobile GPU and CPU can be lifted once entered the performance mode. To achieve the maximum performance, all experiments conducted on mobile platform are performed under such setup.

VI. DESIGN AND CONSTRUCTION OF SOFTWARE SYSTEM

A. Purpose

Due to the reason mentioned in the introduction section, only Nvidia and AMD GPU are supported in the Tensorflow framework. For the purpose of accelerating it on mobile GPU, OpenCL code was embedded into the Tensorflow MatMul kernel to by-pass the original program execution routine and routed it to the OpenCL-accelerated version. Originally, the computation workload was handled by Eigen library, which is a cutting-edge C++ linear algebra library.

The matrix multiplication operation was identified as system bottleneck in most AI training or inference tasks [4]. Thus, a dedicated class called ` *clMatMulEngine*` was designed to handled the floating point (FP32) matrix multiplication and off-load the computation from mobile CPU to mobile GPU via OpenCL code.

B. Design challenges

Perhaps the most demanding part of the code injection challenge is understanding the Tensorflow core codebase due to the lack of internal documentation revealed by Google and all the high-level C++ template representation. Converting matrices data from the `Eigen::Tensor` to OpenCL style `*cl_mem*` object required some understandings of both framework.

Moreover, the matrix multiplications is originally handled by a single line of code `*out.device(d) = in0.contract(in1, dim_pair);*`, which hides lots of implementation details. For instance, the object called `*dim_pair*` contains the info for whether two matrices should be transposed before the actual multiplication. Shortage of such implementation details left the development process struggling. In normal computation, matrices won't be transposed and the bug is invisible. During the AI model training process, matrices will be transposed if multiple multiplication operations are performed sequentially. As a result, two weeks were spent on the identification of this problem because the un-transposed computed results lead to problematic gradient calculation and a silent failure on training task.

C. Architecture

As the complexity of this project grows, experiments with higher code complexity have to be supported by the original design of `*clMatMulEngine*`. Experimenting with new OpenCL optimization technique sometimes requires modification in host-side code. For instance, applying new OpenCL memory optimization techniques often introduces additional OpenCL event objects to handle the program synchronization or a new memory flag for OpenCL memory object. With object oriented programming (OOP) in mind, the software architecture is defined as follow.

The original `*clMatMulEngine*` C++ class became the parent class of all child subclasses. It handles all common OpenCL operations such as host-side OpenCL initialization, common debugging functions etc. Common member variables in were included such as the size of matrices, OpenCL context object, OpenCL device object, OpenCL command queue object. Moreover, Virtual C++ functions were added to define the behavior of inherited classes. For instance, a subclass inherited from `*clMatMulEngine*` should implement a `*clEnd()*` function which releases all OpenCL related objects used in the operation and returns `*CL_SUCCESS*` if all functions completed as expected. A `*memLoad()*` function which copies the computed results from an OpenCL memory object to a Tensorflow defined `*Eigen::TensorMap*`. A `*memInit*()` method that reads in two `*Eigen::TensorMap*` objects (two matrices) and replicates the results to OpenCL memory objects.

`*binaryLoaderInterface*` was created for loading compiled OpenCL kernel dynamically, which increased the performance because the compilation process is time-consuming. This interface defined a virtual method `*loadFromBinaryCompute()*` which loads the compiled OpenCL kernel binary into an OpenCL program object and perform the computation. The implementation detail of a virtual function is defined in the inherited classes.

Three child classes of `*clMatMulEngine*` are as follow. A `clQualcommFP32Engine` class inherits from `*clMatMulEngine*` class and `*binaryLoaderInterface*` class, it's designed to handle floating point 32 bits matrix multiplication on Qualcomm Adreno GPU and loads the compiled OpenCL kernel binary at runtime. `*clQualcommFP16Engine*` was created with similar functionality compared to `*clQualcommFP32Engine*`. The difference lies in the memory copying operation because the floating point of 16 bits is half of the size of 32 bits. Also, different kernel functions are called. `*clBLASTEngine*` is a child class of `*clMatMulEngine*`. The reason for creating such class is to investigate the performance of the CLBlast open source OpenCL BLAS (Basic Linear Algebra Subprogram) library. The source code for `*clMatMulEngine*` can be found in *Appendix A: clMatMulEngine design source code.*

D. Workflow

The workflow of `*clMatMulEngine*` is no different from other OpenCL applications. It involves the following operations, host-side code initialization, memory copy to OpenCL devices, submit OpenCL kernel, memory copy from OpenCL devices back to host, release OpenCL objects. In the scope of the `*clMatMulEngine*` design, `*hostInit()*` function does the host-side initialization, `*memInit()*` handles memory copy to OpenCL devices, `*loadFromBinaryCompute()*` submits OpenCL kernels, `*memLoad()*` copies results back to the host, `*clEnd()*` releases OpenCL objects. The implementation details are included in the appendix.

E. GPU optimization technique applied in clMatMulEngine

OpenCL adheres to a relaxed memory model, some parts of the memory consistency issues are implementation specific and the details are left to device vendor. For Adreno mobile GPU, some optimization techniques are discussed in the official Qualcomm OpenCL programming guide [11] that are specific to Adreno GPU. Adequate techniques are applied in the design of `*clMatMulEngine*` and will be discussed in this section. The discussion will be separated into two parts, memory optimization and binary kernel optimization respectively.

Memory optimization technique is important on mobile devices because of several limitations on mobile platforms. There're limited system RAM on mobile devices and training AI applications is ofter memory consuming because it usually happens in batch. Loading a batch of data into the training program is more efficient and is a widely applied method. In this project, out of memory situation was observed when dealing with large batch size. Discovered from the resource monitor, the training application filled up the memory and resulted in a system freeze. In details, memory copy is required in an OpenCL application to copy the existing data from host to an OpenCL device; however, the mobile GPU shares the last level memory with CPU, which makes the operation unnecessary because the original and the copied memory all sit in the system RAM. During the training process, an OpenCL accelerated Tensroflow MatMul functor will make a copy of the existing Tensorflow allocated data and consumed twice the size of a normal batch.

The solution to the problem mentioned above is the zero copy method mentioned in the programming guide [11]. An OpenCL memory object can be created without introducing unnecessary copy if the flag `*CL_MEM_ALLOC_HOST_PTR*` is used plus the usage of OpenCL memory map function instead of memory copy one. The `*clMatMulEngine*` is free from out of memory problem with this memory optimization technique applied.

The binary kernel optimization technique is important because an OpenCL object is created in the Tensorflow framework for each MatMul operation. (It's not the best practice of doing so and will be discussed in the limitation section.) For each MatMul operation, the kernel code should be compiled in order to create an OpenCL kernel object. Discovered from the Snapdragon profiler, the compilation time for OpenCL kernel is time consuming and slow down the performance dramatically because there're plenty of MatMul operations in an AI application. With this observation in mind, a simple OpenCL device compiler was created to support offline compilation, which pre-compiles the kernel and load the compiled binary at runtime. The `*clCreateProgramWithSource*` function was replaced with `*clCreateProgramWithBinary*` to create an OpenCL program object in the `*clMatMulEngine*`.

VII.EXPERIMENT RESULTS

A. Experiment — CLBlast evaluation

The CLBlast library is an open source OpenCL BLAS library [19]. 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 13.

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 14.

A.3.Tensorflow overhead

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 15.

A.4.Problem encountered

Mentioned in the previous section, a class `*clBLASTEngine*` was created to incorporate the library itself into the Tensorflow framework. Introducing a new library into the Tensorlfow framework is complicated by the usage of Bazel

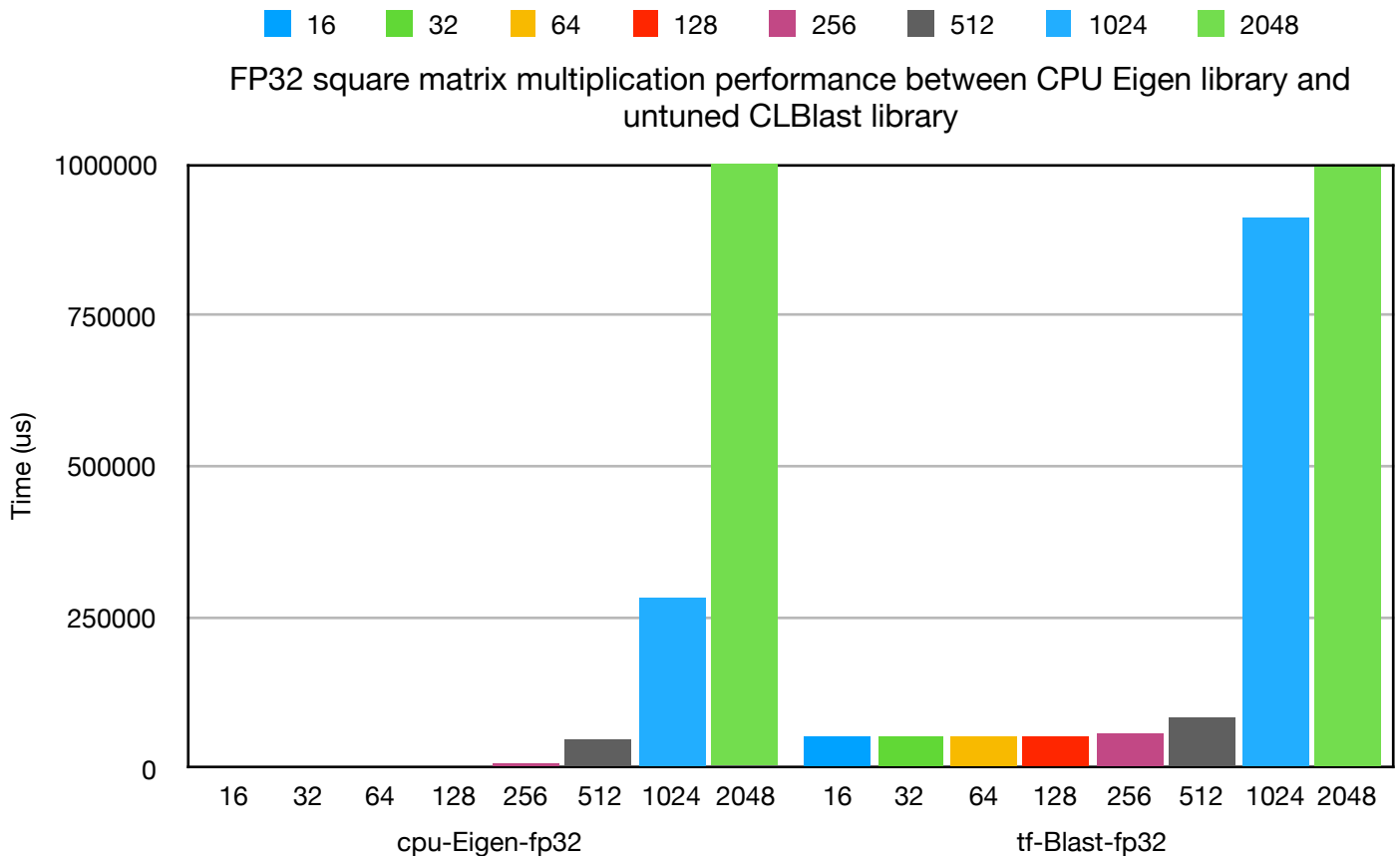

Figure 13. The FP32 square matrix multiplication performance between CPU Eigen library and untuned CLBlast library. Different colors show matrices of different size. Y-axis is the time needed in microsecond (us).

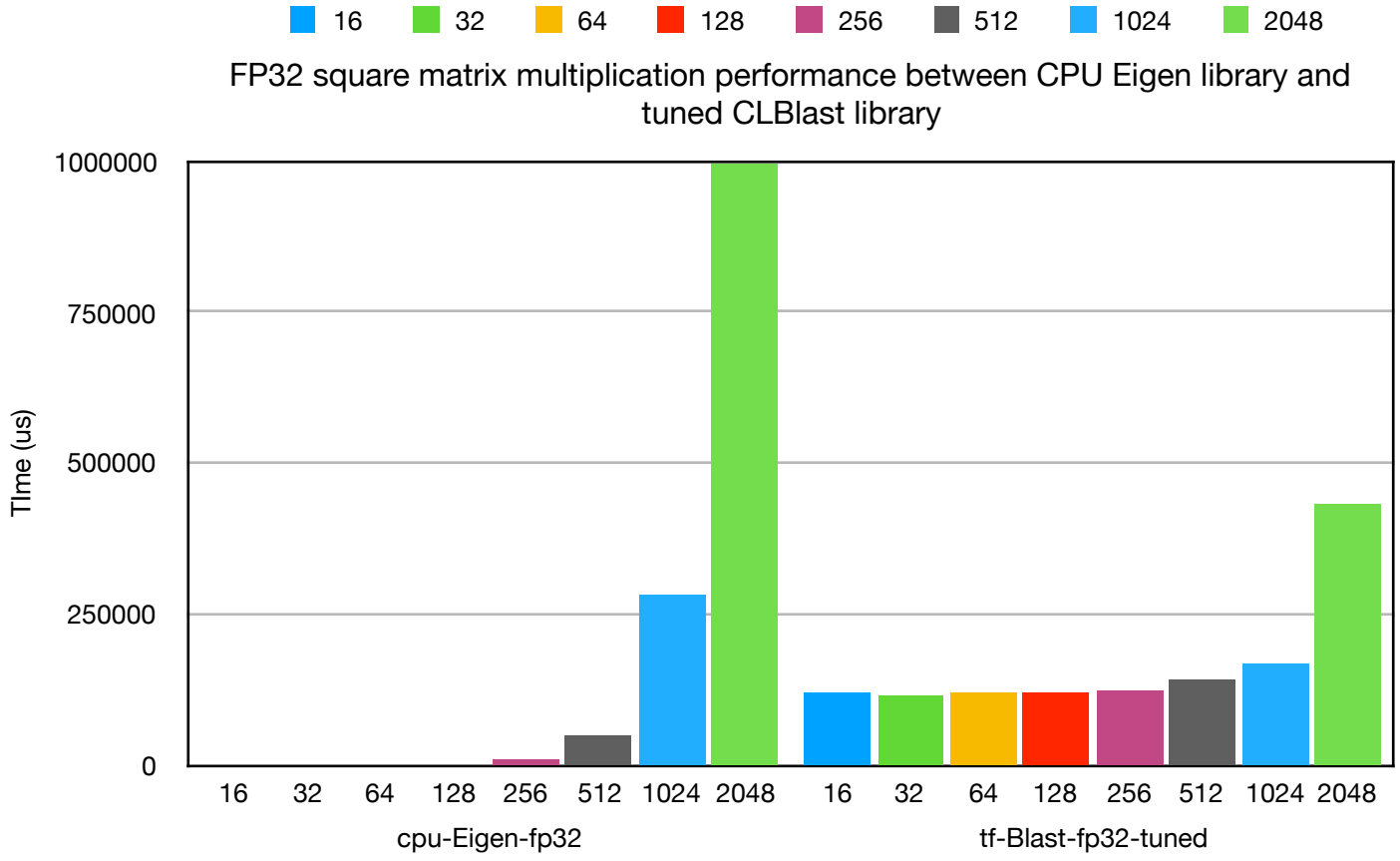

Figure 14. The FP32 square matrix multiplication performance between CPU Eigen library and tuned CLBlast library. Different colors show matrices of different size. Y-axis is the time needed in microsecond (us).

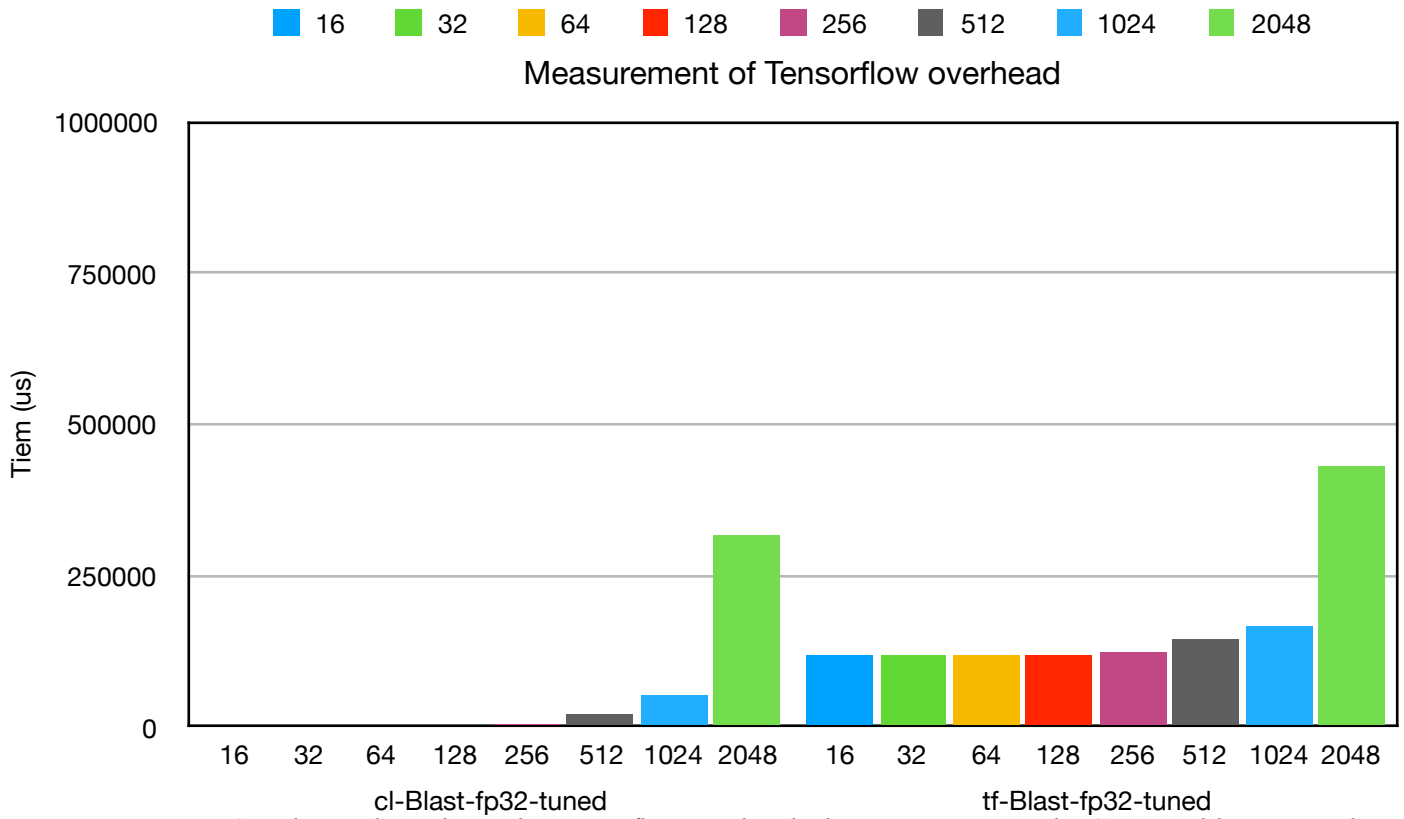

Figure 15. This is chart shows the Tensorflow overhead when incorporating the CLBLast library into the framework. Different colors represent matrices of different size. Y-axis is the square matrix multiplication time in microsecond (us). Left section is the CLBlast program running as a normal OpenCL program. Right section is the performance of incorporating it into the Tensorflow framework.

building system. A `*native.new_http_archive*` object should be added the `*workspace.blz*` file in the Tensorflow repository. It should contain the URL link to the source code of the repository, the SHA256 sum of the downloaded source code, and a Bazel BUILD file that defines the compilation rules for external libraries.

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 — Tensorflow MatMul test

The purpose of this experiment is to verify the correctness of `*clMatMulEngine*` and identify the amount of overhead added to the Tensorflow framework. A Tensorflow program `*opencl-matmul*` was created with C++ API from scratch in this experiment.

`*opencl-matmul*` test was designed as follow. On the desktop platform, two `*tf.placeholder*` python objects were created using Tensorflow Python API and passed to `*tf.matmul*` matrix multiplication ops for calculation. The overall computational graph was stored using `*tf.train.write_graph*` API. On the Android platform, the program rebuilds a computational graph by reading the stored .pb file. Two Tensorflow 2D Tensors were created and initialized with small random floating points values. Both were loaded into the runtime by the function call `*session->Run()*`, which also retuned the computed value in another Tensor object. To verify the correctness of the computation, two `*Eigen::Matrix*` objects were created and initialized by the same value used in the initialization of Tensors. Eventually, the matrix multiplication results handled by the Eigen framework, which depends on mobile CPU, was compared with the one computed by Tensorflow framework in a element-wise manner. For each element in the multiplied matrix, absolute error was accumulated and averaged to show the overall differences.

Various options are available for this test including whether the matrices should be transposed before multiplication, the number of iteration to run the test, and the size of the matrices. This experiment was used to load various kinds of OpenCL MatMul kernels in the next section. The source code can be found in *Appendix F: Tensorflow MatMul test — opencl-matmul source code.*

C. 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 mentioned in previous section.

C.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

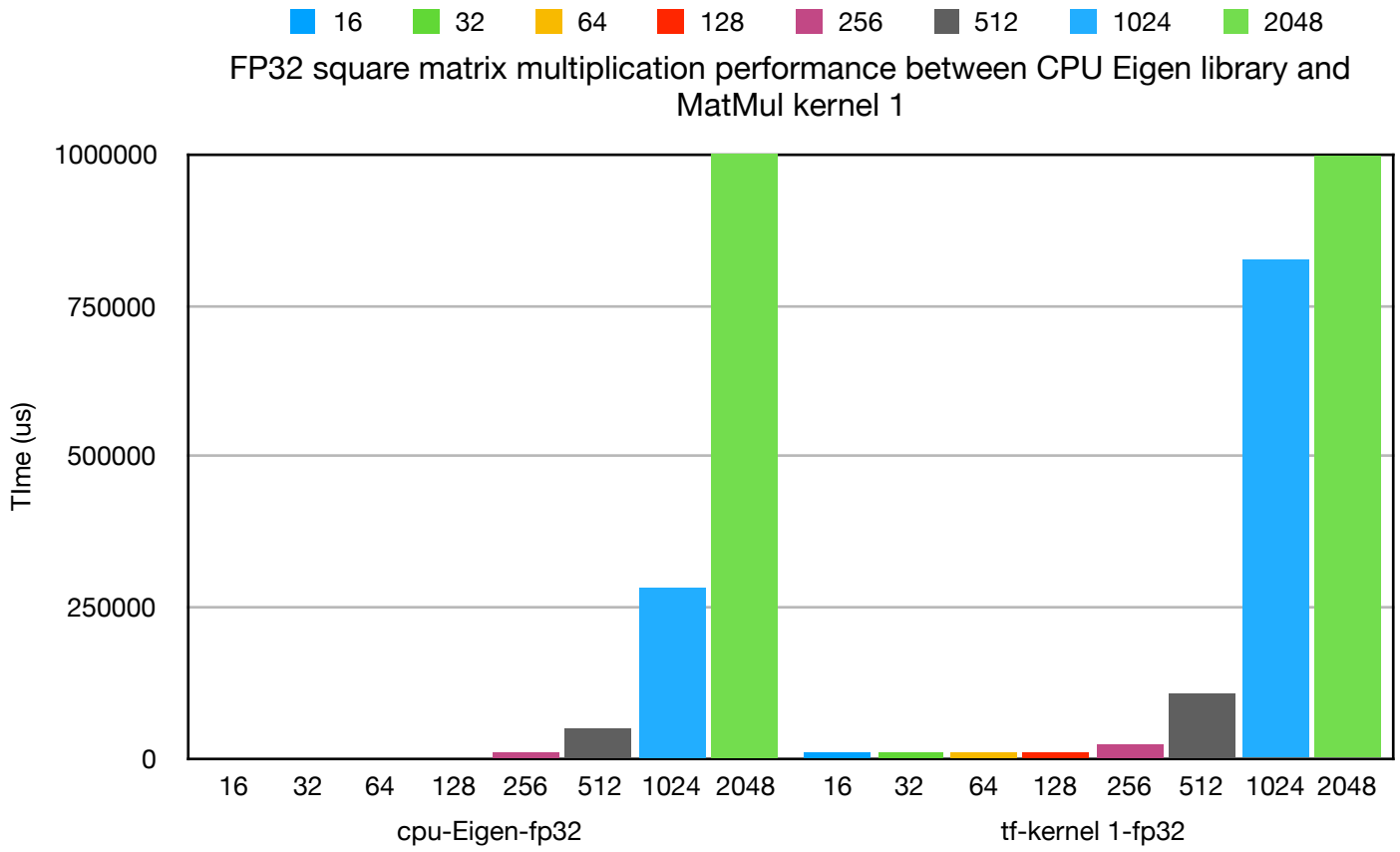

Figure 16. The FP32 square matrix multiplication performance between CPU Eigen library and OpenCL kernel version 1. Different colors show matrices of different size. Y-axis is the time needed in microsecond (us).

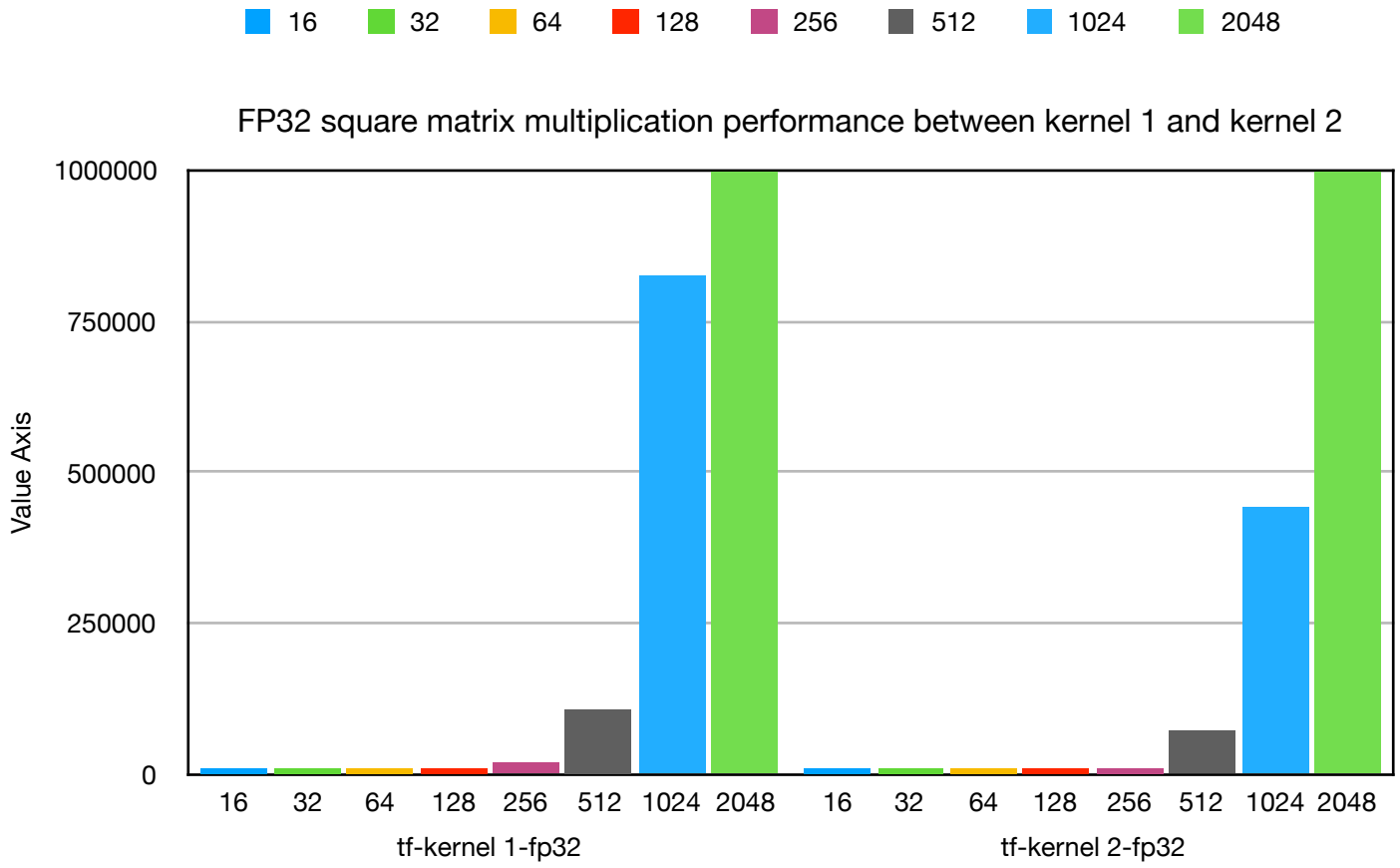

Figure 17. The FP32 square matrix multiplication performance between OpenCL kernel version 1 and OpenCL kernel version 2. Different colors show matrices of different size. Y-axis is the time needed in microsecond (us).

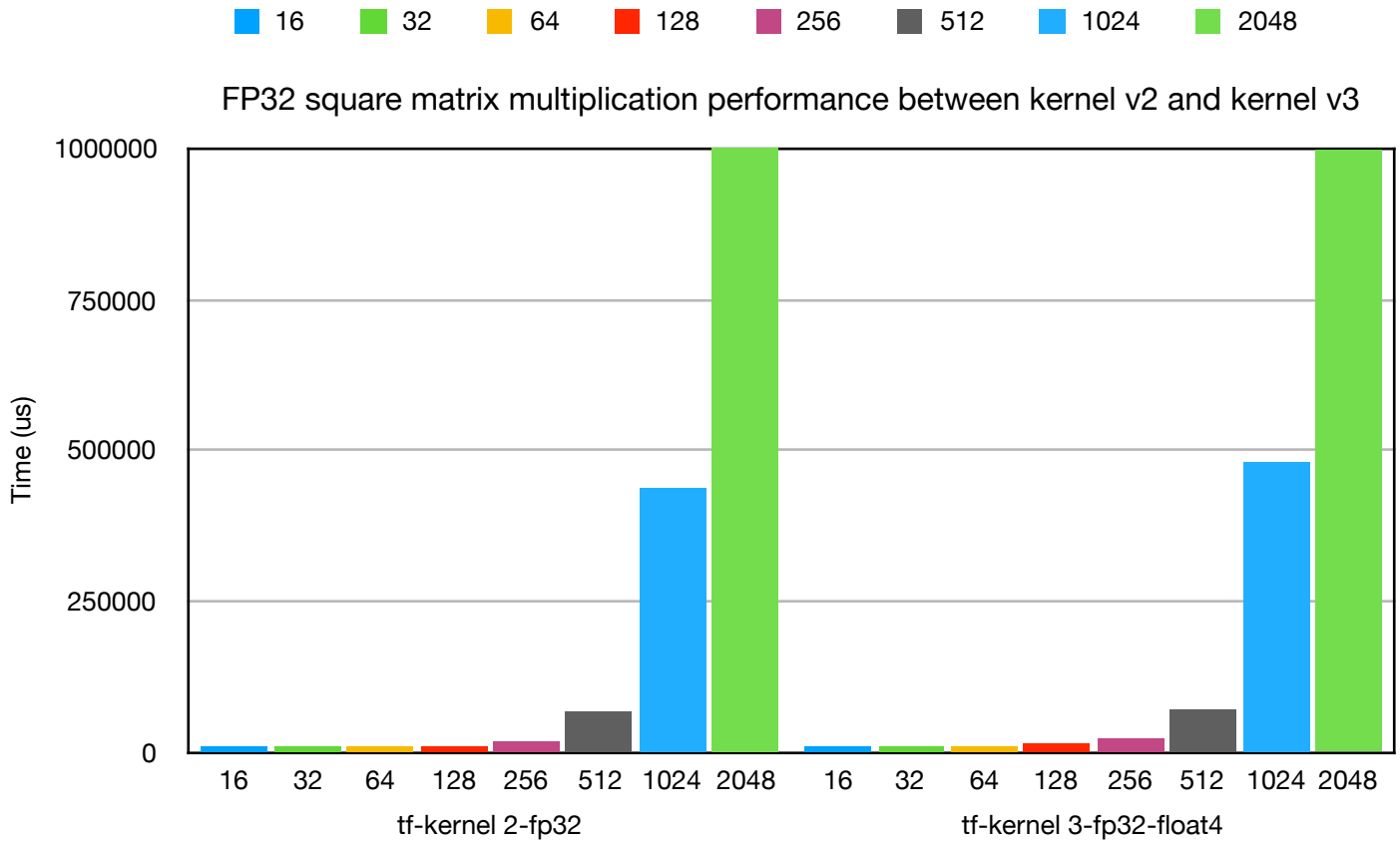

Figure 18. The FP32 square matrix multiplication performance between OpenCL kernel version 2 and OpenCL kernel version 3. Different colors show matrices of different size. Y-axis is the time needed in microsecond (us).

in Figure 16. Kernel version 1 is considered as the baseline performance because it's the most straightforward matrix multiplication kernel available. The programming logic 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.

C.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 17. 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.

The limitation of such kernel is the capability to handle boundary cases. The local memory size if fixed to 16*16 in this kernel, which gives miscalculated the results given matrix size isn't a multiple of 16.

C.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 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 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

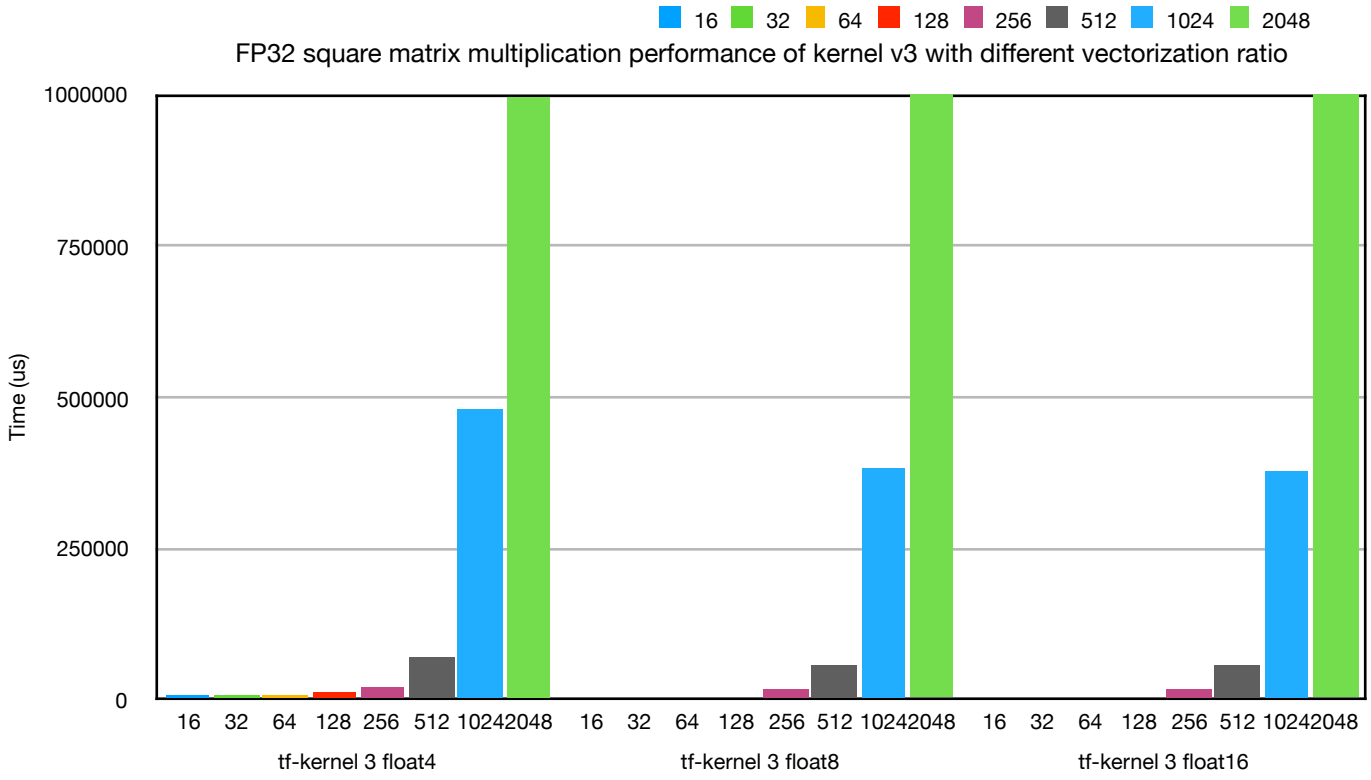

Figure 19. The FP32 square matrix multiplication performance of OpenCL kernel version 3 with different vectorization ratio. Different colors show matrices of different size. Y-axis is the time needed in microsecond (us).

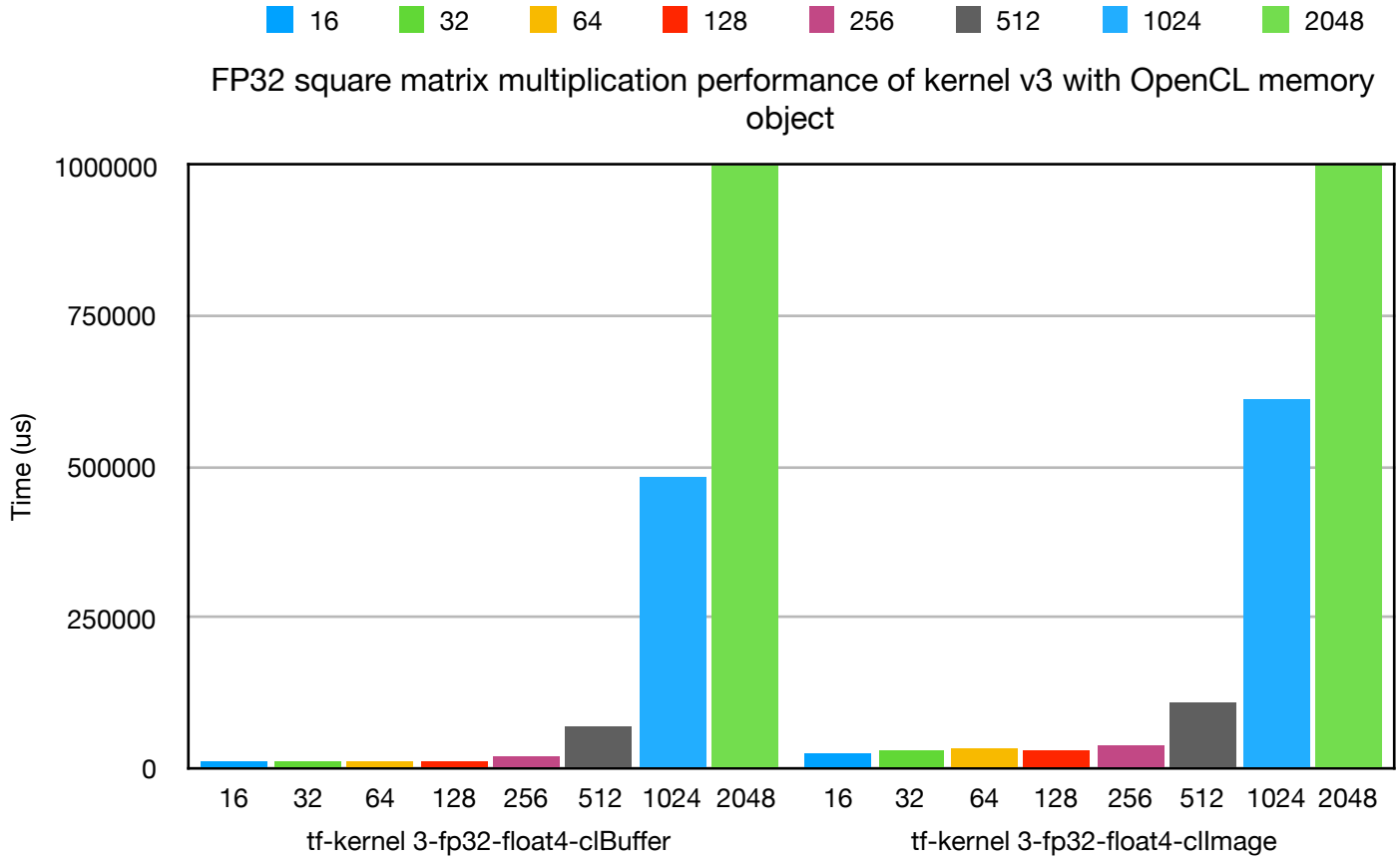

Figure 20.5. The FP32 square matrix multiplication performance of OpenCL kernel version 3 with different OpenCL memory object. Different colors show matrices of different size. Y-axis is the time needed in microsecond (us).

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 18. Out-of-expectation, the new kernel is worse than kernel 2.

C.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 observe the best ratio. Results shown on Figure 19. The best vectorization ratio float datatype for this kernel is 16.

C.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 2, 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 2. The optimal work group size for MatMul kernel 3 with different vectorization ratio.

C.6.Different OpenCL memory object

This optimization technique is specific to Adreno GPU because of its GPU architecture mentioned in the analyze of the problem section or best illustrated in Figure 2. The optimization trick was mentioned in a blog post on Qualcomm developer network [20]. 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

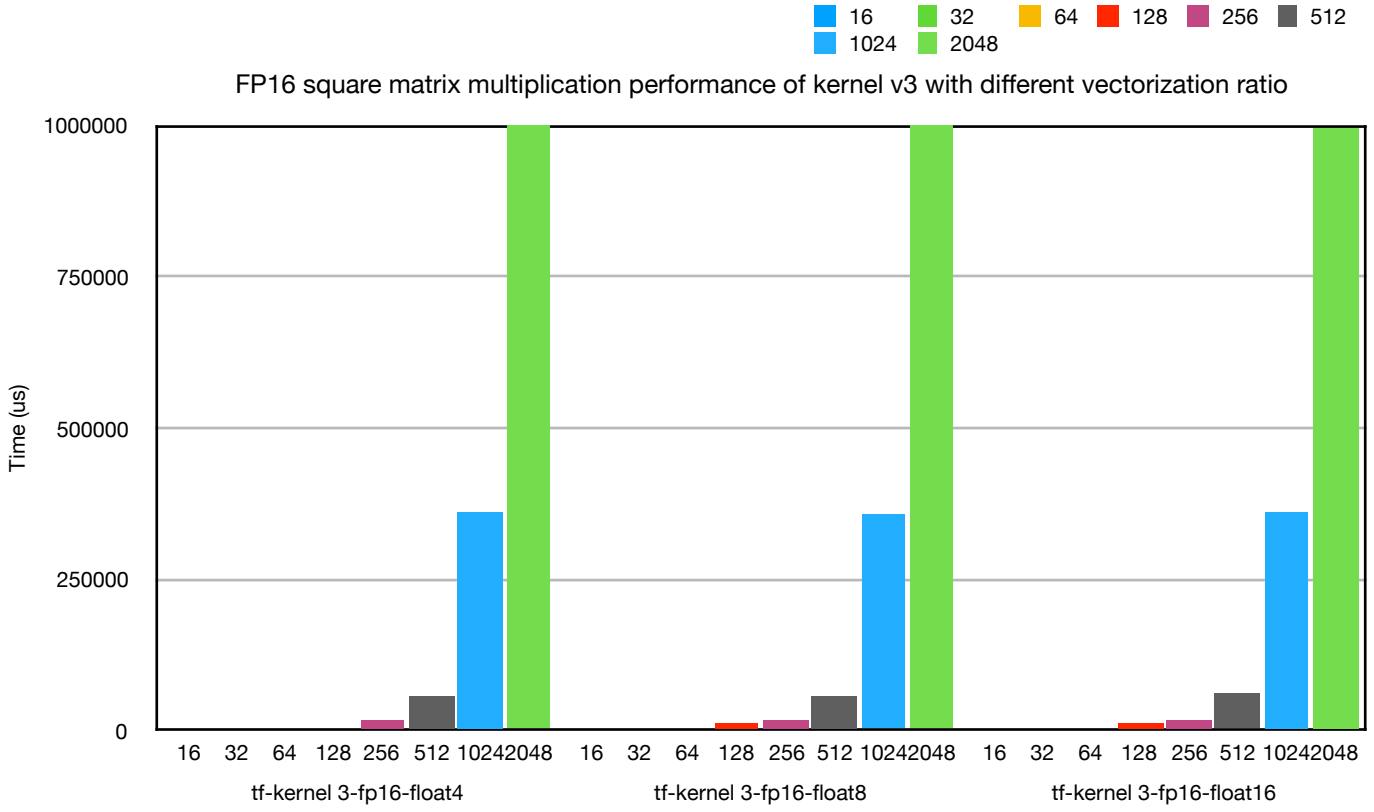

Figure 20. The FP16 square matrix multiplication performance of OpenCL kernel version 3 with different vectorization ratio. Different colors show matrices of different size. Y-axis is the time needed in microsecond (us).

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 20.5.

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.

C.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 20.

On Figure 20, there's minor improvement in speed but the quality deteriorated as the number of matrices grow. 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.

C.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, and loop unrolling.

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.

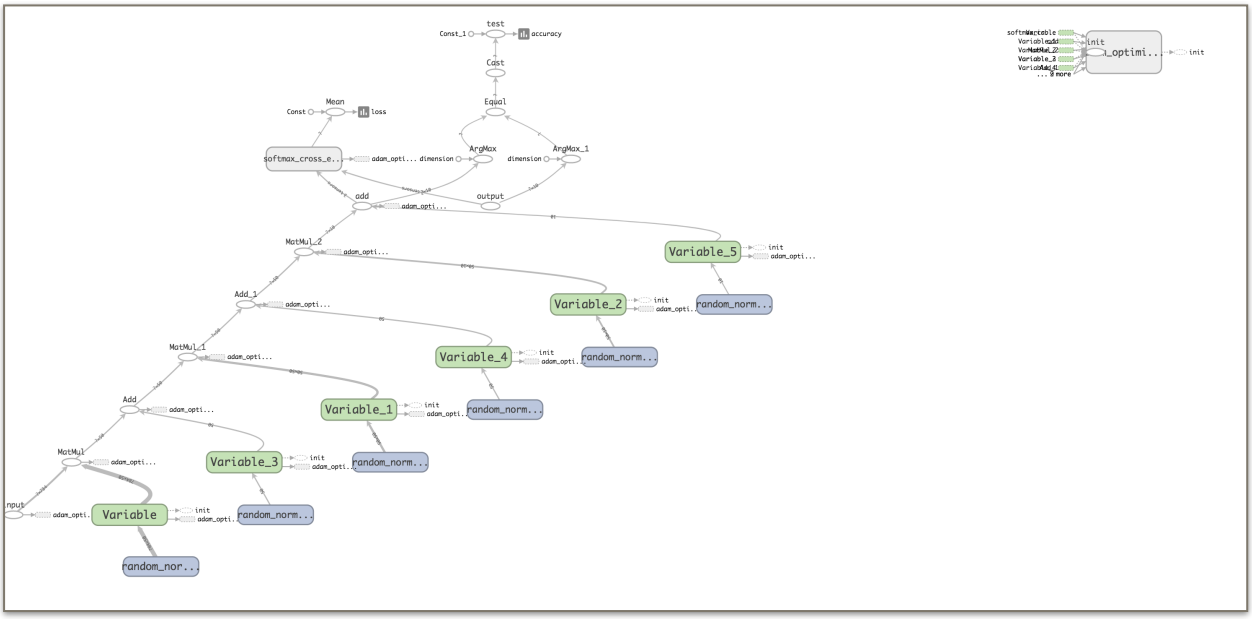

Figure 21. The computational graph for MLP model.

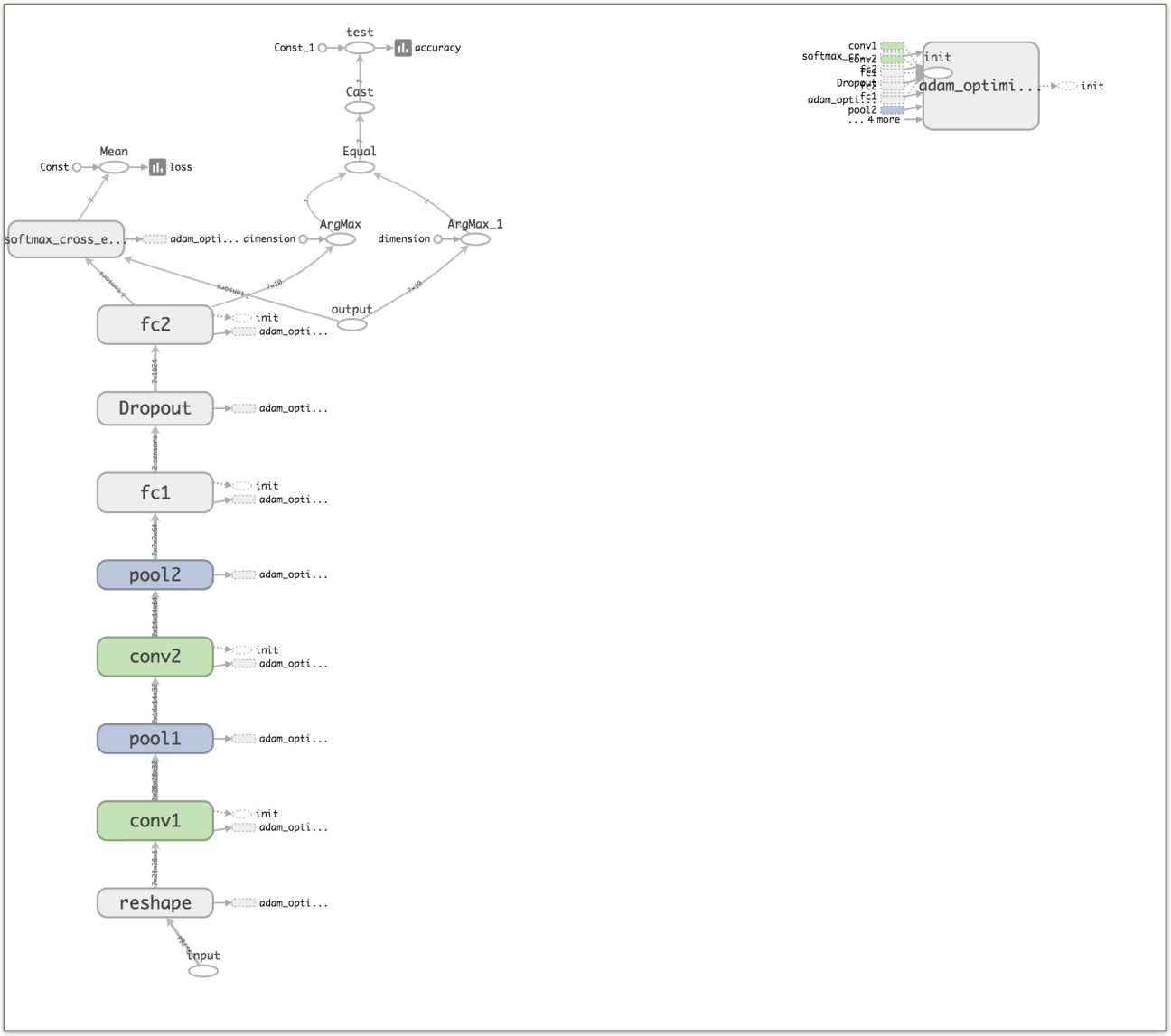

Figure 22. The computational graph for DNN model.

Loop unrolling is a common optimization technique to reduce the number branch instruction in a loop. This is applied to MatMul kernel by adding a compiler hint (#progma unroll) before a loop. A compiler will unroll the loop if it predicts such operation will increase the performance. However, loop unrolling gives no obvious improvement in this case.

D. 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.

D.1.AI model structure

In this experiment, two AI model, MLP and DNN, will be trained. The structure of two AI models are defined by the following Tensorflow computational graph. The structure of MLP model is shown on Figure 21, and the DNN model shown on Figure 22. Refer to the analyze of problem section to revisit analyze of MatMul operation in each AI model.

D.2.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. The source of this pure training program can be found in *Appendix C: MNIST pure trainer program source code.*

D.3.The design of training logger program

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

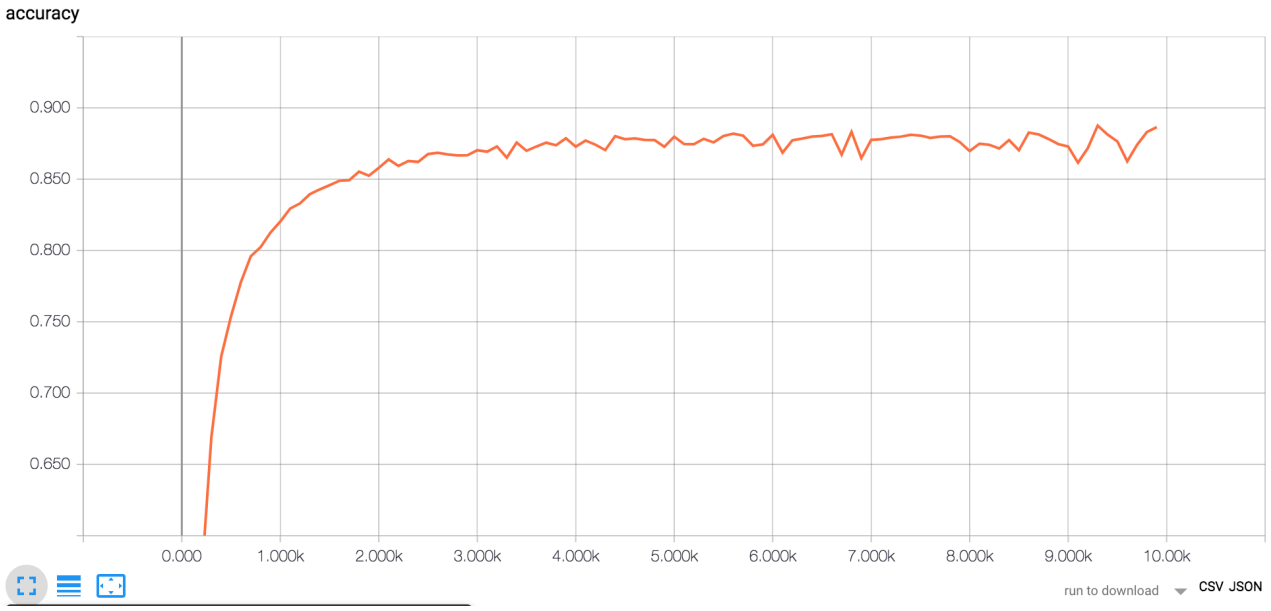

Figure 23. The MLP training accuracy on desktop computer.

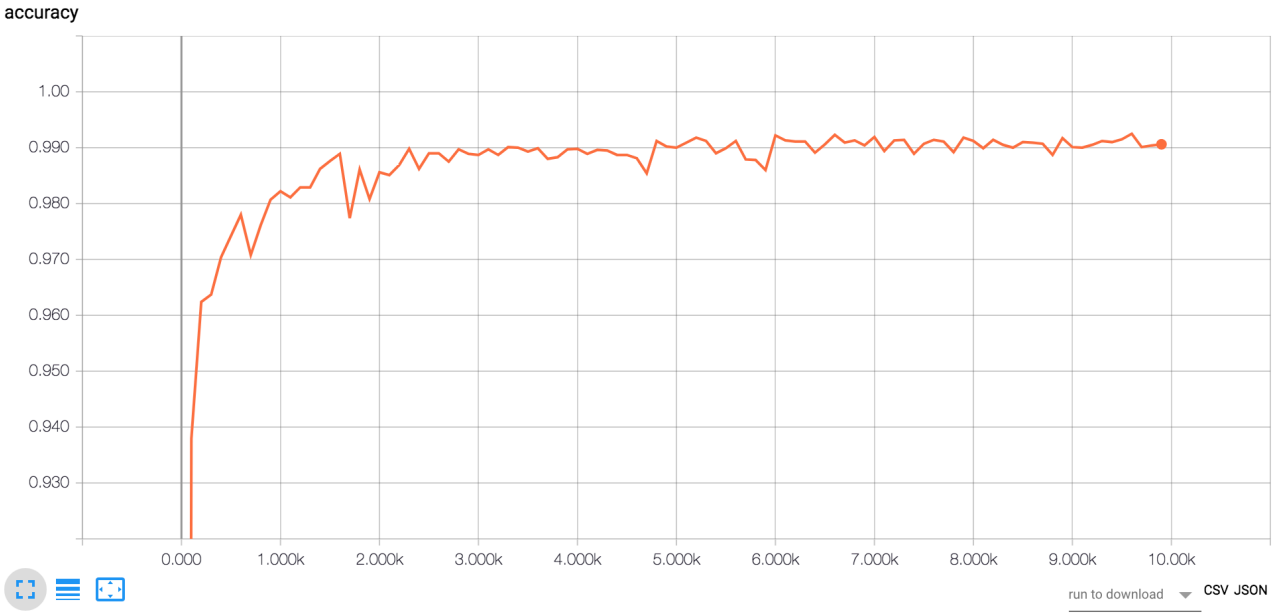

Figure 24. The DNN training accuracy on desktop computer.

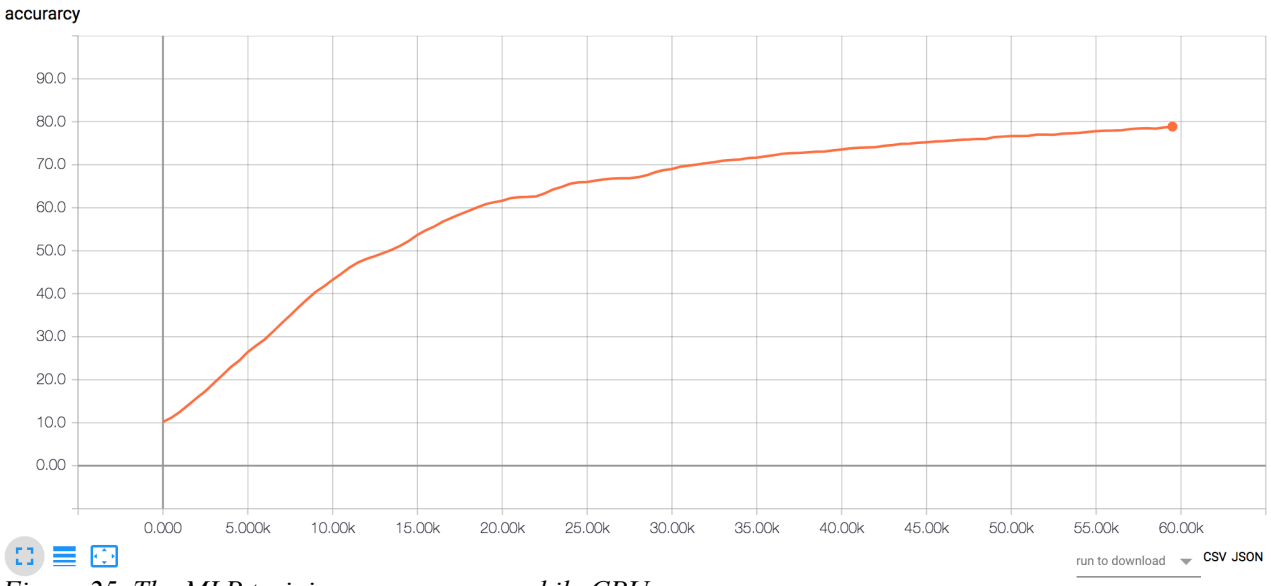

Figure 25. The MLP training accuracy on mobile CPU.

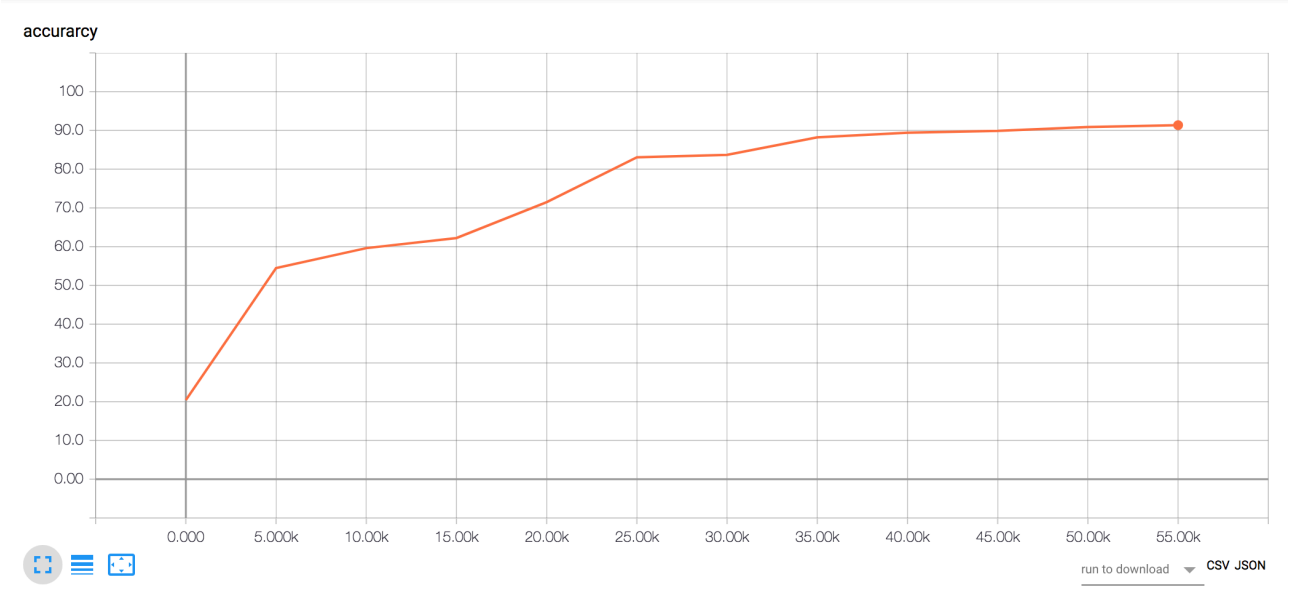

Figure 26. The DNN training accuracy on mobile CPU.

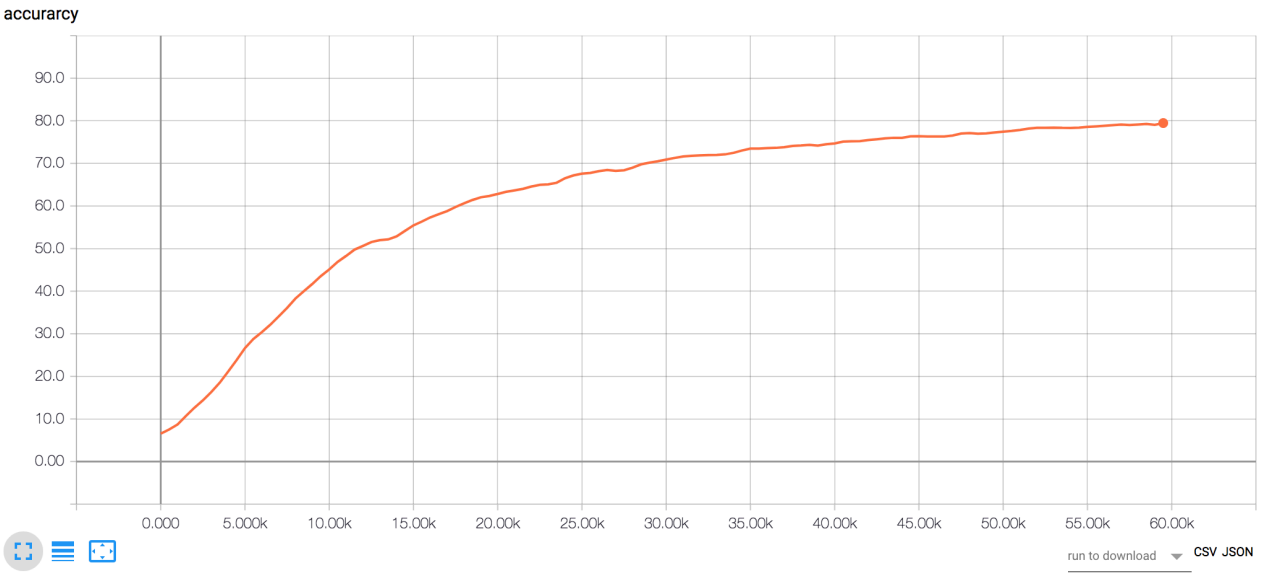

Figure 27. The MLP training accuracy on mobile GPU.

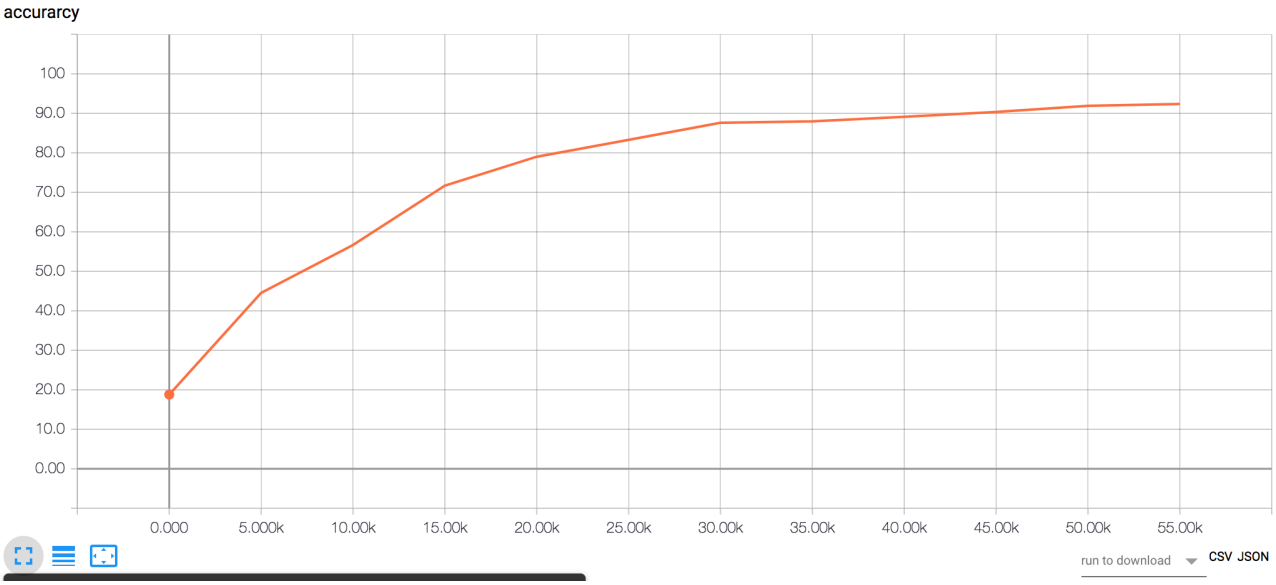

Figure 28. The DNN training accuracy on mobile GPU.

compare with the training progress on mobile CPU/ GPU. The source code can be found in *Appendix D: MNIST training logger program source code.*

D.4. Training accuracy

Figure 23 and Figure 24 show the MLP and DNN model training progress on desktop CPU. Figure 25 (MLP) and Figure 26 (DNN) show the training progress on mobile CPU. Figure 27 (MLP) and Figure 28 (DNN) show the training progress on mobile GPU. From Figure 23 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, and the implementation reaches the project goal.

D.5. 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 time decreased by merely 6%. Figure 29. visualizes the results of training time on mobile GPU.

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

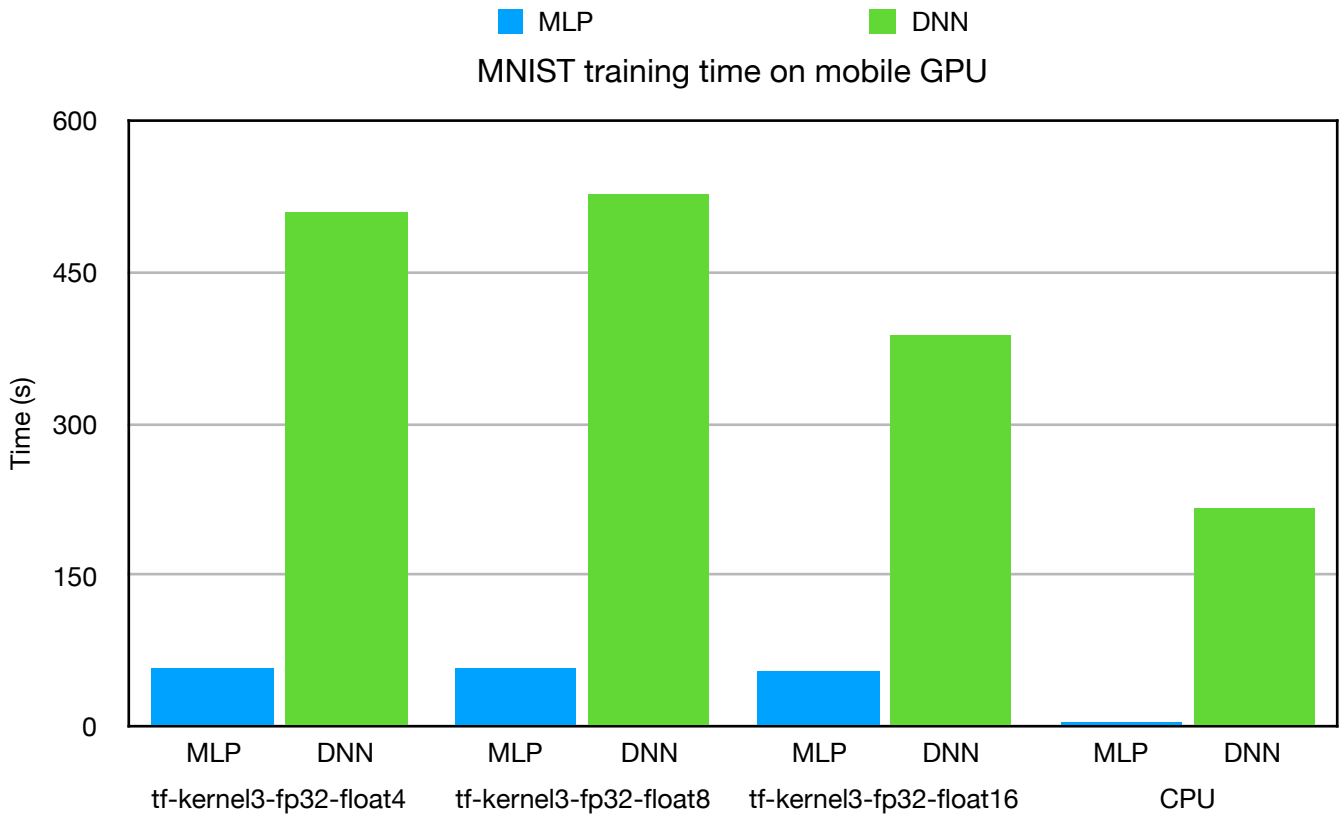

Figure 29. The MNIST training performance. Blue bar represents the time for MLP, and green for DNN. Y-axis is the time needed in second .

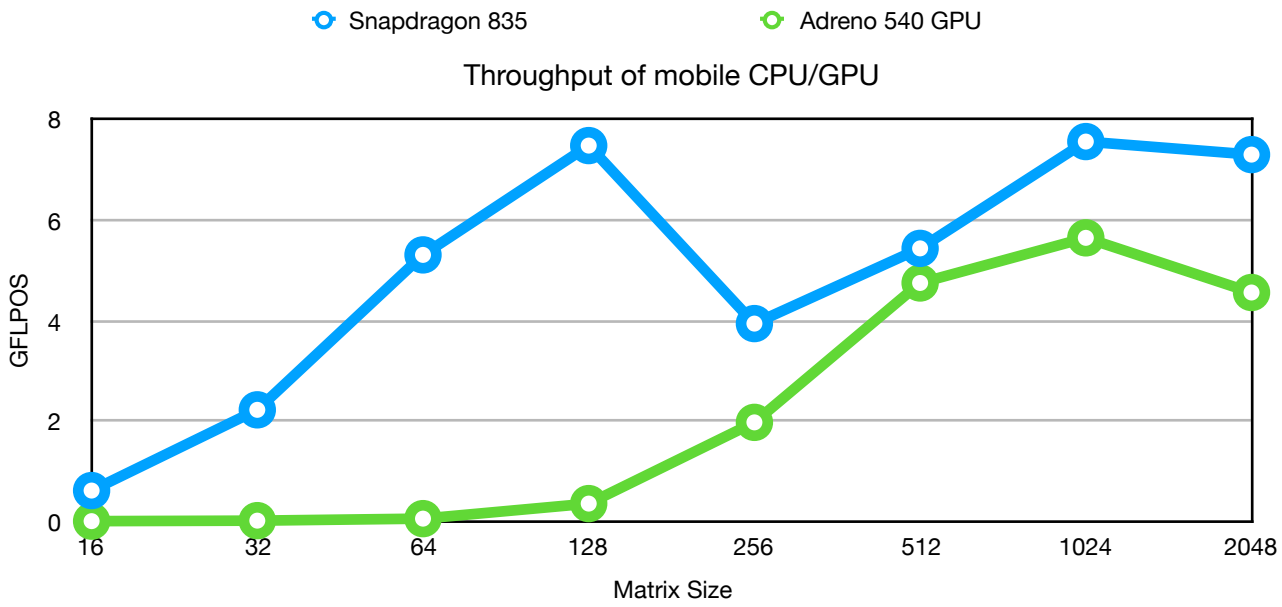

Figure 30. The throughput of Snapdragon 835 CPU and Adreno 540 GPU in square matrix multiplication task. X-axis is the size of the matrix. Y-axis is the throughput in GFLOPS.

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

E. 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 [21], 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 30. The result obtained is slightly lower than GeekBench's measurement, the maximum throughput for CPU is ~7 GFLOPS, and GPU is ~ 6 GFLOPS.

VIII. 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 14. 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 as shown in Figure 31. Since the performance measurements in Figure 14. 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 15. 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 float16 data type is the

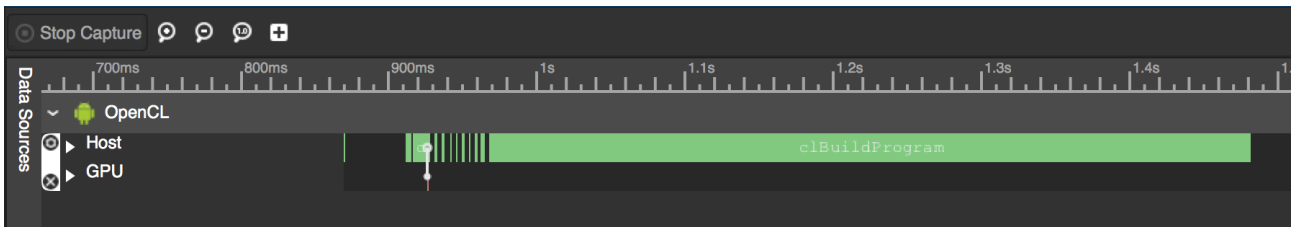

Figure 31. The time spent on building CLBlast OpenCL kernel.

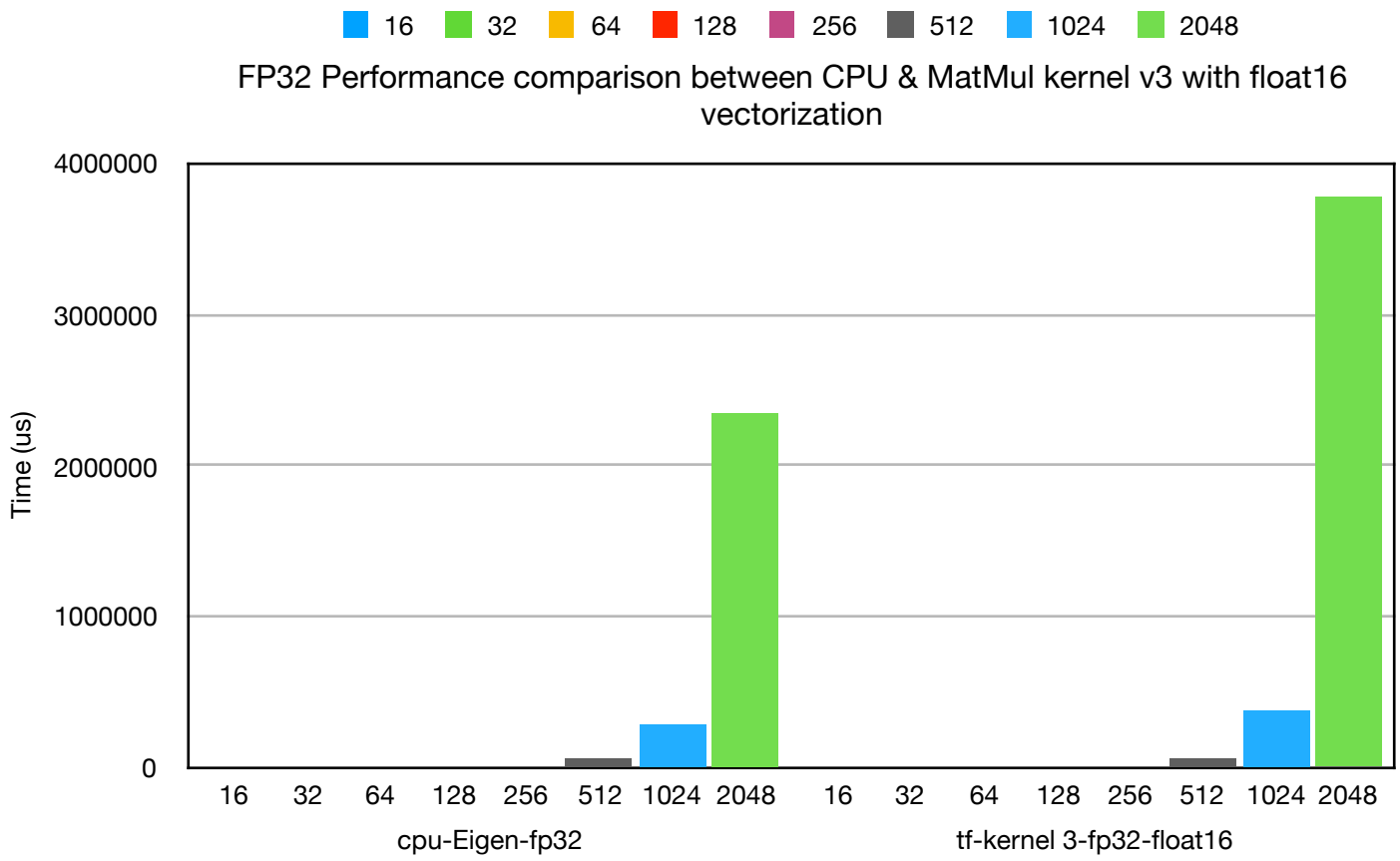

Figure 32. The FP32 square matrix multiplication performance of OpenCL kernel version 3 with float16 vectorization and CPU. Different colors show matrices of different size. Y-axis is the time needed in microsecond (us).

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 32. 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.

IX.LIMITATION

The limitations of this project 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.

X. 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.

XI.REFERENCE

1. 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.
2. Tensorflow. *Tensorflow Lite*. Available from: https://www.tensorflow.org/mobile/tflite/.
3. Tensorflow. *Tensorflow Architecture*. Available from: https://www.tensorflow.org/extend/architecture.
4. Jia, Y., Learning semantic image representations at a large scale. 2014: University of California, Berkeley.
5. Software, C. *ComputeCpp*. Available from: https://github.com/codeplaysoftware/computecpp-sdk.
6. *triSYCL*. Available from: https://github.com/triSYCL/triSYCL.
7. Žužek, P., Implementacija knjiˇznice SYCL za heterogeno raˇcunanje. 2016.
8. Žužek, P. *sycl-gtx Library*. 2016; Available from: https://github.com/ProGTX/sycl-gtx.
9. Qualcomm. *Tensorflow optimized for Snapdragon Hexagon 682 DSP*. Available from: https://www.qualcomm.com/news/onq/2017/01/09/tensorflow-machine-learning-now-optimized-snapdragon-835-and-hexagon-682-dsp.
10. Qualcomm. *Snapdragon Neural Processing Engine*. Available from: https://developer.qualcomm.com/software/snapdragon-neural-processing-engine.
11. Qualcomm Snapdragon(TM) Mobile Platform OpenCL General Programming and Optimization. 2017.
12. ARM. *ARM Mali GPU OpenCL Driver Developer Guide 3.2*. Available from: https://developer.arm.com/docs/100614/latest/introduction/about-opencl.
13. Qualcomm. *Snapdragon Profiler*. Available from: https://developer.qualcomm.com/software/snapdragon-profiler.
14. Khronos. *The OpenCL Specification*. 2018; Available from: https://www.khronos.org/registry/OpenCL/specs/opencl-2.2.html.
15. McVoy, L.W. and C. Staelin. lmbench: Portable Tools for Performance Analysis. in USENIX annual technical conference. 1996. San Diego, CA, USA.
16. 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.
17. wichtounet. *Simple C++ reader for MNIST dataset*. 2018; Available from: https://github.com/wichtounet/mnist.
18. *Tensorboar logger*. Available from: https://github.com/RustingSword/tensorboard_logger.
19. Nugteren, C., *CLBLast: A tuned openCL BLAS library.* arXiv preprint arXiv:1705.05249, 2017.
20. Qualcomm. *Matrix Multiply on Adreno GPUs*. 2016; Available from: https://developer.qualcomm.com/blog/matrix-multiply-adreno-gpus-part-1-opencl-optimization.
21. GeekBench. *The Qualcomm Snapdragon 835 Performance Preview*. Available from: https://www.anandtech.com/show/11201/qualcomm-snapdragon-835-performance-preview/2.

XII.APPENDIX

A. clMatMulEngine design source code

```
// clMatMulEngine<float>
//     |
//     v
// clQualcommFP32Engine <---- binaryLoaderInterface

// clMatMulEngine<float>
//     |
//     v
// clQualcommFP16Engine <---- binaryLoaderInterface

// clMatMulEngine<float>
//     |
//     v
// clBLASTEngine

#ifndef MATMUL_CL_FUNCTOR_H_
#define MATMUL_CL_FUNCTOR_H_

#include <fstream>

#include "third_party/eigen3/unsupported/Eigen/CXX11/Tensor"
#include "tensorflow/core/framework/tensor.h"
#include "tensorflow/core/framework/tensor_types.h"
#include "tensorflow/core/lib/hash/hash.h"
#include "tensorflow/core/platform/logging.h"

#define CL_USE_DEPRECATED_OPENCL_1_2_APIS // to disable deprecation warnings

// Includes the CLBlast library (C interface)
#include "clblast_c.h"

////////////////////////////////////////////////////////////////////////////////
// OpenCL status checker
#define CL_CHECK(_expr)                                                \
 {                                                                     \
   cl_int _err = _expr;                                                \
   if( _err != CL_SUCCESS) {                                           \
     std::cerr << "OpenCL Error: " << #_expr << " returned " << (int)_err     \
     << std::endl;                                                     \
   }                                                                   \
 }
// OpenCL return type checker
#define CL_CHECK_ERR(_expr)                                            \
 ({                                                                    \
   cl_int _err = CL_INVALID_VALUE;                                     \
   decltype(_expr) _ret = _expr;                                       \
   if (_err != CL_SUCCESS) {                                           \
     std::cerr << "OpenCL Error: " << #_expr << " returned " << (int)_err     \
     << std::endl;                                                     \
   }                                                                   \
   _ret;                                                               \
 })

////////////////////////////////////////////////////////////////////////////////
```

```
// float to cl_half conversions
#ifndef INFINITY
  #define INFINITY 1.0/0.0
#endif

#ifndef NAN
  #define NAN 0.0/0.0
#endif

typedef union {
  int32_t i;
  float f;
} FloatConvUnion;

cl_half float_to_cl_half(float value){

  FloatConvUnion u;
  u.f = value;
  cl_half half = (u.i >> 16) & 0x8000; // sign
  cl_half fraction = (u.i >> 12) & 0x007ff; // fraction with extra bit for rounding
  cl_half exponent = (u.i >> 23)  & 0xff; // exponent

  if(exponent < 0x0067) // Return signed zero if zero or value is too small for denormal half
    return half;

  if(exponent > 0x008e){// value was NaN or Inf
    half |= 0x7c00u; // Make into inf
    half |= exponent == 255 && (u.i & 0x007fffffu); // If value was NaN make this into NaN
    return half;
  }

  if(exponent < 0x0071){// Denormal
    fraction |= 0x0800u;

    // rounding
    half |= (fraction >> (0x0072 - exponent)) + ((fraction >> (0x0071 - exponent)) & 1);
    return half;
  }

  half |= ((exponent - 0x0070) << 10) | (fraction >> 1);
  half += fraction & 1;// rounding
  return half;
}

////////////////////////////////////////////////////////////////////////////////
// clSetKernelArg Helper
#define SET_GEMM_TN_KERNEL_ARG(M, K, N, clMemA, clMemB, clMemC, localSize,
localMemType, \
  iter)                                                                 \
  CL_CHECK( clSetKernelArg(clGemmKernel, 0, sizeof(cl_ushort), &M) );          \
  CL_CHECK( clSetKernelArg(clGemmKernel, 1, sizeof(cl_ushort), &K) );          \
  CL_CHECK( clSetKernelArg(clGemmKernel, 2, sizeof(cl_ushort), &N) );          \
  CL_CHECK( clSetKernelArg(clGemmKernel, 3, sizeof(cl_mem), &clMemA) );           \
  CL_CHECK( clSetKernelArg(clGemmKernel, 4, sizeof(cl_mem), &clMemB) );           \
```

```
  CL_CHECK( clSetKernelArg(clGemmKernel, 5, sizeof(cl_mem), &clMemC) );                \
  CL_CHECK( clSetKernelArg(clGemmKernel, 6, localSize * sizeof(localMemType), NULL) );  \
  CL_CHECK( clSetKernelArg(clGemmKernel, 7, sizeof(cl_ushort), &iter) );               \

////////////////////////////////////////////////////////////////////////////////
// clSetKernelArg Helper
#define SET_TRANS_KERNEL_ARG(ROW, COL, clMem, clMem_T, iter)                       \
  CL_CHECK( clSetKernelArg(clTransKernel, 0, sizeof(cl_ushort), &ROW) );           \
  CL_CHECK( clSetKernelArg(clTransKernel, 1, sizeof(cl_ushort), &COL) );           \
  CL_CHECK( clSetKernelArg(clTransKernel, 2, sizeof(cl_mem), &clMem) );            \
  CL_CHECK( clSetKernelArg(clTransKernel, 3, sizeof(cl_mem), &clMem_T) );          \
  CL_CHECK( clSetKernelArg(clTransKernel, 4, sizeof(cl_ushort), &iter) );          \

using namespace std;

namespace tensorflow {
  typedef Eigen::ThreadPoolDevice CPUDevice;

  // clMatMulEngine abstract class (interface), computing datatype T
  template<class T> class clMatMulEngine {
    public:

    // Concrete methods
      // clMatMulEngine initializaiotn function
      cl_int hostInit(
        typename functor::MatMulTypes<T>::in_type in0,
        typename functor::MatMulTypes<T>::in_type in1,
        typename functor::MatMulTypes<T>::out_type out,
        const Eigen::array<Eigen::IndexPair<Eigen::DenseIndex>, 1>& dim_pair)
      {
        // Matrix dimension init
        RowA = in0.dimension(0);
        ColA = in0.dimension(1);
        RowB = in1.dimension(0);
        ColB = in1.dimension(1);
        RowC = out.dimension(0);
        ColC = out.dimension(1);

        // Matrix size checking
        int matrixSizeLimit = 0xffff; // Maximum value for cl_ushort
        if( RowA > matrixSizeLimit ||
            ColA > matrixSizeLimit ||
            RowB > matrixSizeLimit ||
            ColB > matrixSizeLimit ||
            RowC > matrixSizeLimit ||
            ColC > matrixSizeLimit )
        {
          LOG(ERROR) << "Matrix of Size Larger than " << matrixSizeLimit <<
          " isn't supported";
        }

        // Matrix size init
        a_size = sizeof(T) * RowA * ColA;
        b_size = sizeof(T) * RowB * ColB;
```

```
    c_size = sizeof(T) * RowC * ColC;

    // Matrix transpose
    a_traspose = ( dim_pair[0].first == 0 ) ? true : false;
    b_traspose = ( dim_pair[0].second == 1 ) ? true : false;

    // Query platforms
    CL_CHECK( clGetPlatformIDs(1, &platform, NULL) );

    // Query devices
    CL_CHECK( clGetDeviceIDs(platform, CL_DEVICE_TYPE_ALL, 1, &clDevice, NULL) );

    // Create context
    clCtx = CL_CHECK_ERR( clCreateContext(NULL, 1, &clDevice, NULL, NULL, &_err) );

    // Create command clQueue
    clQueue = CL_CHECK_ERR( clCreateCommandQueue(clCtx, clDevice, 0, &_err) );

    return CL_SUCCESS;
  }

  // Print debug info
  void debug( bool print=true ){
    if( print ){
      LOG(INFO) << "Dealing with datatype of size " << sizeof(T);
      LOG(INFO) << "MatrixA = [" << RowA << "," << ColA  << "]";
      LOG(INFO) << "MatrixB = [" << RowB << "," << ColB  << "]";
      LOG(INFO) << "MatrixC = [" << RowC << "," << ColC  << "]";
    }
  }

  // Print input-output matrices
  void printMatrix(
    typename functor::MatMulTypes<T>::in_type in0,
    typename functor::MatMulTypes<T>::in_type in1,
    typename functor::MatMulTypes<T>::out_type out)
  {
    LOG(INFO) << "MatMul Matrix details";
    LOG(INFO) << std::endl << in0;
    LOG(INFO) << std::endl << in1;
    LOG(INFO) << std::endl << out;
  }

// Virtual methods
  // Release all OpenCL related resourcse
  virtual cl_int clEnd() = 0;

  // Load computed results back to memroy
  virtual cl_int memLoad(typename functor::MatMulTypes<T>::out_type out) = 0;

  // OpenCL memeory object init
  virtual cl_int memInit( typename functor::MatMulTypes<T>::in_type in0,
    typename functor::MatMulTypes<T>::in_type in1) = 0;
```

```
protected:

  // Default matrix dimension
  size_t RowA = 0;
  size_t ColA = 0;
  size_t RowB = 0;
  size_t ColB = 0;
  size_t RowC = 0;
  size_t ColC = 0;

  // Matrix tranpose info
  bool a_traspose;
  bool b_traspose;

  // Default matrix size
  size_t a_size = 0;
  size_t b_size = 0;
  size_t c_size = 0;

  // OpenCL host side object
  cl_platform_id platform;
  cl_device_id clDevice;
  cl_context clCtx;
  cl_command_queue clQueue;

  // Timer
  std::chrono::high_resolution_clock::time_point timer;

  void startTimer(){
    timer = std::chrono::high_resolution_clock::now();
  }

  double read_us(){
    auto elapsed_time = std::chrono::high_resolution_clock::now() - timer;
    return std::chrono::duration<double, std::micro>(elapsed_time).count();
  }

  // Performance calculator
  void getPerformance(){

    std::ofstream ofs ("performance.log", std::ios_base::app);

    long double delta_t = read_us() * 1e-6; // delta_t in second

    double bandwidth = 1e-9*(a_size+b_size+c_size)/delta_t;
    ofs << bandwidth << " GB/s, ";

    long double gflops = 1e-9*(RowA*ColA*RowB*ColB*RowC*ColC)/delta_t;
    ofs << gflops << " GFLOPS\n";

    ofs.close();
  }

};  // class clMatMulEngine
```

```
// binaryLoaderInterface abstract class (interface)
class binaryLoaderInterface{
  public:

  // Virtual method
    // Compile & Compute the results
    virtual cl_int loadFromBinaryCompute() = 0;

  protected:

  // Concrete methods
    // Read OpenCL binary file from disk
    int read_file(unsigned char **output, size_t *size, const char *name)
    {
      FILE* fp = fopen(name, "rb");
      if (!fp) {
        LOG(ERROR) << "Fail to read cl kernel binary " << std::string( name );
        return -1;
      }

      fseek(fp, 0, SEEK_END);
      *size = ftell(fp);
      fseek(fp, 0, SEEK_SET);

      *output = (unsigned char *)malloc(*size);
      if (!*output) {
        fclose(fp);
        return -1;
      }
      fread(*output, *size, 1, fp);
      fclose(fp);
      return 0;
    }

    // Show clKernel object info
    void debugOpenclKernel(cl_kernel cl_kernel, cl_device_id cl_device){

      // Kernel info
      size_t wgSize = 0;
      size_t compiledWgSize[3];
      cl_ulong localMemSize = 0;
      size_t perfHint;
      cl_ulong privateMemSize = 0;

      CL_CHECK( clGetKernelWorkGroupInfo(cl_kernel, cl_device,
                CL_KERNEL_WORK_GROUP_SIZE, sizeof(size_t), &wgSize, NULL) );
      CL_CHECK( clGetKernelWorkGroupInfo(cl_kernel, cl_device,
                CL_KERNEL_COMPILE_WORK_GROUP_SIZE, 3 * sizeof(size_t),
                &compiledWgSize, NULL) );
      CL_CHECK( clGetKernelWorkGroupInfo(cl_kernel, cl_device,
                CL_KERNEL_LOCAL_MEM_SIZE, sizeof(cl_ulong), &localMemSize,
                NULL) );
      CL_CHECK( clGetKernelWorkGroupInfo(cl_kernel, cl_device,
```

```
// binaryLoaderInterface abstract class (interface)
class binaryLoaderInterface{
  public:

  // Virtual method
    // Compile & Compute the results
    virtual cl_int loadFromBinaryCompute() = 0;

  protected:

  // Concrete methods
    // Read OpenCL binary file from disk
    int read_file(unsigned char **output, size_t *size, const char *name)
    {
      FILE* fp = fopen(name, "rb");
      if (!fp) {
        LOG(ERROR) << "Fail to read cl kernel binary " << std::string( name );
        return -1;
      }

      fseek(fp, 0, SEEK_END);
      *size = ftell(fp);
      fseek(fp, 0, SEEK_SET);

      *output = (unsigned char *)malloc(*size);
      if (!*output) {
        fclose(fp);
        return -1;
      }
      fread(*output, *size, 1, fp);
      fclose(fp);
      return 0;
    }

    // Show clKernel object info
    void debugOpenclKernel(cl_kernel cl_kernel, cl_device_id cl_device){

      // Kernel info
      size_t wgSize = 0;
      size_t compiledWgSize[3];
      cl_ulong localMemSize = 0;
      size_t perfHint;
      cl_ulong privateMemSize = 0;

      CL_CHECK( clGetKernelWorkGroupInfo(cl_kernel, cl_device,
                CL_KERNEL_WORK_GROUP_SIZE, sizeof(size_t), &wgSize, NULL) );
      CL_CHECK( clGetKernelWorkGroupInfo(cl_kernel, cl_device,
                CL_KERNEL_COMPILE_WORK_GROUP_SIZE, 3 * sizeof(size_t),
                &compiledWgSize, NULL) );
      CL_CHECK( clGetKernelWorkGroupInfo(cl_kernel, cl_device,
                CL_KERNEL_LOCAL_MEM_SIZE, sizeof(cl_ulong), &localMemSize,
                NULL) );
      CL_CHECK( clGetKernelWorkGroupInfo(cl_kernel, cl_device,
```

```
    // Return CL_SUCCESS if all resources are released successfully
    return CL_SUCCESS;
  }

  cl_int memLoad(typename functor::MatMulTypes<float>::out_type out){

    // Use the map function to return clBufferA pointer to the host <= blocking
    clHostPtrC = ( cl_float * ) clEnqueueMapBuffer(clQueue, clBufferC, CL_TRUE,
                 CL_MAP_READ, 0, c_size, 0, NULL, NULL, NULL);

    // Read computed result back to host
    for( auto idx = 0 ; idx < RowC*ColC ; idx++){
      out.data()[idx] = clHostPtrC[idx];
    }

    // Release OpenCL resources
    CL_CHECK( clEnd() );

    // Return if the results are loaded to memory & OpenCL resources are released
    return CL_SUCCESS;
  }

  cl_int memInit(
    typename functor::MatMulTypes<float>::in_type in0,
    typename functor::MatMulTypes<float>::in_type in1)
  {

    // Use zero copy to avoid additional memeory copy
    // Matrix A
    clBufferA = clCreateBuffer(clCtx, CL_MEM_HOST_WRITE_ONLY |
CL_MEM_ALLOC_HOST_PTR,
                a_size, NULL, NULL);
    // Use the map function to return clBufferA pointer to the host <= non-blocking
    clHostPtrA = ( cl_float * ) clEnqueueMapBuffer(clQueue, clBufferA, CL_FALSE,
                          CL_MAP_WRITE, 0, a_size, 0, NULL,
                          &mapBufferEvents[0], NULL);
    // Matrix B
    clBufferB = clCreateBuffer(clCtx, CL_MEM_HOST_WRITE_ONLY |
CL_MEM_ALLOC_HOST_PTR,
                b_size, NULL, NULL);
    // Use the map function to return clBufferA pointer to the host <= non-blocking
    clHostPtrB = ( cl_float * ) clEnqueueMapBuffer(clQueue, clBufferB, CL_FALSE,
                          CL_MAP_WRITE, 0, b_size, 0, NULL,
                          &mapBufferEvents[1], NULL);

    // Create GPU buffer for transposed matrices only if needed
    if( a_traspose ){
      clBufferA_T = clCreateBuffer(clCtx, CL_MEM_HOST_NO_ACCESS, a_size, NULL,
NULL);
    }
    if( !b_traspose ){
      clBufferB_T = clCreateBuffer(clCtx, CL_MEM_HOST_NO_ACCESS, b_size, NULL,
NULL);
```

```
        }

        // Wait for completion
        CL_CHECK( clWaitForEvents(2, mapBufferEvents) );

        // Host update the buffer using pointer clHostPtrA in host address space
        for( auto idx = 0 ; idx < RowA*ColA ; idx ++){
          clHostPtrA[ idx ] = in0.data()[idx];
        }
        // Host update the buffer using pointer clHostPtrB in host address space
        for( auto idx = 0 ; idx < RowB*ColB ; idx ++){
          clHostPtrB[ idx ] = in1.data()[idx];
        }

        // Unmap the object -> Used in the OpenCL kernel
        CL_CHECK( clEnqueueUnmapMemObject( clQueue, clBufferA, (void*) clHostPtrA,
                0, NULL, &unMapBufferEvents[0] ) );

        // Unmap the object -> Used in the OpenCL kernel
        CL_CHECK( clEnqueueUnmapMemObject( clQueue, clBufferB, (void*) clHostPtrB,
                0, NULL, &unMapBufferEvents[1] ) );

        // Matrix C
        clBufferC = clCreateBuffer(clCtx, CL_MEM_HOST_READ_ONLY |
CL_MEM_ALLOC_HOST_PTR,
                    c_size, NULL, NULL);

        // Wait for completion
        CL_CHECK( clWaitForEvents(2, unMapBufferEvents) );
        return CL_SUCCESS;
      }

      cl_int loadFromBinaryCompute()
      {

        unsigned char* clKernelBinaryFile = NULL;
        size_t clKernelBinSize = 0;
        // Read compiled OpenCL kernel binary file from disk
        read_file(&clKernelBinaryFile, &clKernelBinSize, "matmul.bin" );

        // Create an OpenCL program object from binary
        clProgram = CL_CHECK_ERR( clCreateProgramWithBinary(clCtx, 1, &clDevice,
                            &clKernelBinSize,
                            (const unsigned char **)&clKernelBinaryFile,
                            NULL, &_err) );

        // OpenCL build program
        CL_CHECK( clBuildProgram(clProgram, 1, &clDevice, "-cl-fast-relaxed-math" , NULL,
NULL) );

        // Create OpenCL GEMM kernel object
        // clGemmKernel = CL_CHECK_ERR( clCreateKernel(clProgram,
"MatMul_TN_1D_Fp32_Float4" , &_err) );
```

```
        // clGemmKernel = CL_CHECK_ERR( clCreateKernel(clProgram,
"MatMul_TN_1D_Fp32_Float8" , &_err) );
        clGemmKernel = CL_CHECK_ERR( clCreateKernel(clProgram,
"MatMul_TN_1D_Fp32_Float16" , &_err) );

        // Create OpenCL Transpose kernel object
        // clTransKernel = CL_CHECK_ERR( clCreateKernel(clProgram,
"MatTrans_1D_Fp32_Float4" , &_err) );
        // clTransKernel = CL_CHECK_ERR( clCreateKernel(clProgram,
"MatTrans_1D_Fp32_Float8" , &_err) );
        clTransKernel = CL_CHECK_ERR( clCreateKernel(clProgram,
"MatTrans_1D_Fp32_Float16" , &_err) );

        cl_ushort gemmKernelIter;
        cl_ushort transKernelIter;

        startTimer();

        // Handle Matrices Transpose
        if( a_traspose && b_traspose ){ // Transpose A: yes, Transpose B: yes

          transKernelIter = ColA >> 4;
          gemmKernelIter = RowA >> 4;

          // Transpose A
          SET_TRANS_KERNEL_ARG(RowA, ColA, clBufferA, clBufferA_T, transKernelIter );

          CL_CHECK( clEnqueueNDRangeKernel(clQueue, clTransKernel, 1, NULL,
                  &RowA, NULL, 0, NULL, &transKernelEvent[0]) );

          SET_GEMM_TN_KERNEL_ARG(ColA, RowA, RowB, clBufferA_T, clBufferB,
            clBufferC, ColA, float, gemmKernelIter );

          const size_t global = ColA;
          CL_CHECK( clEnqueueNDRangeKernel(clQueue, clGemmKernel, 1, NULL,
                  &global, NULL, 1, transKernelEvent, &gemmKernelEvent) );

          CL_CHECK( clWaitForEvents(1, &gemmKernelEvent) );

        }else if( a_traspose && !b_traspose ){ // Transpose A: yes, Transpose B: no

          transKernelIter = ColA >> 4;
          gemmKernelIter = RowA >> 4;

          // Transpose A
          SET_TRANS_KERNEL_ARG(RowA, ColA, clBufferA, clBufferA_T, transKernelIter );

          CL_CHECK( clEnqueueNDRangeKernel(clQueue, clTransKernel, 1, NULL,
                  &RowA, NULL, 0, NULL, &transKernelEvent[0]) );

          transKernelIter = ColB >> 4;

          // Transpose B
          SET_TRANS_KERNEL_ARG(RowB, ColB, clBufferB, clBufferB_T, transKernelIter );
```

```
    CL_CHECK( clEnqueueNDRangeKernel(clQueue, clTransKernel, 1, NULL,
            &RowB, NULL, 0, NULL, &transKernelEvent[1]) );

    SET_GEMM_TN_KERNEL_ARG(ColA, RowA, ColB, clBufferA_T, clBufferB_T,
      clBufferC, RowA, float, gemmKernelIter );

    const size_t global = ColA;
    CL_CHECK( clEnqueueNDRangeKernel(clQueue, clGemmKernel, 1, NULL,
            &global, NULL, 2, transKernelEvent, &gemmKernelEvent) );

    CL_CHECK( clWaitForEvents(1, &gemmKernelEvent) );

   }else if( !a_traspose && b_traspose ){ // Transpose A: no, Transpose B: yes

    gemmKernelIter = ColA >> 4;

    SET_GEMM_TN_KERNEL_ARG(RowA, ColA, RowB, clBufferA, clBufferB,
      clBufferC, ColA, float, gemmKernelIter );

    const size_t global = RowA;
    CL_CHECK( clEnqueueNDRangeKernel(clQueue, clGemmKernel, 1, NULL,
            &global, NULL, 0, NULL, &gemmKernelEvent) );

    CL_CHECK( clWaitForEvents(1, &gemmKernelEvent) );

   }else if( !a_traspose && !b_traspose ){ // Transpose A: no, Transpose B: no

    transKernelIter = ColB >> 4;
    gemmKernelIter = ColA >> 4;

    // Transpose B
    SET_TRANS_KERNEL_ARG(RowB, ColB, clBufferB, clBufferB_T, transKernelIter );

    CL_CHECK( clEnqueueNDRangeKernel(clQueue, clTransKernel, 1, NULL,
            &ColA, NULL, 0, NULL, &transKernelEvent[0]) );

    SET_GEMM_TN_KERNEL_ARG(RowA, ColA, ColB, clBufferA, clBufferB_T,
      clBufferC, ColA, float, gemmKernelIter);

    const size_t global = RowA;
    CL_CHECK( clEnqueueNDRangeKernel(clQueue, clGemmKernel, 1, NULL,
            &global, NULL, 1, transKernelEvent, &gemmKernelEvent) );

    CL_CHECK( clWaitForEvents(1, &gemmKernelEvent) );
   }

   getPerformance();
   return CL_SUCCESS;
  }

protected:

 // OpenCL memeory object
```

```
    cl_mem clBufferA;
    cl_mem clBufferA_T;
    cl_mem clBufferB;
    cl_mem clBufferB_T;
    cl_mem clBufferC;

    // Host memory data
    cl_float * clHostPtrA;
    cl_float * clHostPtrB;
    cl_float * clHostPtrC;

    // OpenCL events
    cl_event gemmKernelEvent;
    cl_event transKernelEvent[2];
    cl_event mapBufferEvents[2];
    cl_event unMapBufferEvents[2];

    // OpenCL program object
    cl_program clProgram;

    // OpenCL kernel object
    cl_kernel clGemmKernel;
    cl_kernel clTransKernel;

};  // class clQualcommFP32Engine

// clQualcommFP16Engine concrete class using Qualcomm GEMM example
class clQualcommFP16Engine : public clQualcommFP32Engine{
  public:

    cl_int memLoad(typename functor::MatMulTypes<float>::out_type out){

      // Use the map function to return clBufferA pointer to the host <= blocking
      clHostPtrC = ( cl_float * ) clEnqueueMapBuffer(clQueue, clBufferC, CL_TRUE,
                                CL_MAP_READ, 0, c_size, 0, NULL, NULL, NULL);

      // Read computed result back to host
      for( auto idx = 0 ; idx < RowC*ColC ; idx++){
        out.data()[idx] = clHostPtrC[idx];
      }

      // Release OpenCL resources
      CL_CHECK( clEnd() );

      // Return if the results are loaded to memory & OpenCL resources are released
      return CL_SUCCESS;
    }

    cl_int memInit(
      typename functor::MatMulTypes<float>::in_type in0,
      typename functor::MatMulTypes<float>::in_type in1)
    {

      // FP16 is half of the size of FP32
```

```
    a_size = a_size >> 1;
    b_size = b_size >> 1;

    // Use zero copy to avoid memeory copy
    // Matrix A
    clBufferA = clCreateBuffer(clCtx, CL_MEM_HOST_WRITE_ONLY |
CL_MEM_ALLOC_HOST_PTR,
            a_size, NULL, NULL);
    // Use the map function to return clBufferA pointer to the host <= non-blocking
    clHostFp16PtrA = ( cl_half * ) clEnqueueMapBuffer(clQueue, clBufferA, CL_FALSE,
                        CL_MAP_WRITE, 0, a_size, 0, NULL,
                        &mapBufferEvents[0], NULL);
    // Matrix B
    clBufferB = clCreateBuffer(clCtx, CL_MEM_HOST_WRITE_ONLY |
CL_MEM_ALLOC_HOST_PTR,
            b_size, NULL, NULL);
    // Use the map function to return clBufferA pointer to the host <= non-blocking
    clHostFp16PtrB = ( cl_half * ) clEnqueueMapBuffer(clQueue, clBufferB, CL_FALSE,
                        CL_MAP_WRITE, 0, b_size, 0, NULL,
                        &mapBufferEvents[1], NULL);

    // Create GPU buffer for transposed matrices only if needed
    if( a_traspose ){
      clBufferA_T = clCreateBuffer(clCtx, CL_MEM_HOST_NO_ACCESS, a_size, NULL,
NULL);
    }
    if( !b_traspose ){
      clBufferB_T = clCreateBuffer(clCtx, CL_MEM_HOST_NO_ACCESS, b_size, NULL,
NULL);
    }

    // Wait for completion
    CL_CHECK( clWaitForEvents(2, mapBufferEvents) );

    // Host update the buffer using pointer clHostFp16PtrA in host address space
    for( auto idx = 0 ; idx < RowA*ColA ; idx ++){
      clHostFp16PtrA[ idx ] = float_to_cl_half( in0.data()[idx] );
    }
    // Host update the buffer using pointer clHostFp16PtrB in host address space
    for( auto idx = 0 ; idx < RowB*ColB ; idx ++){
      clHostFp16PtrB[ idx ] = float_to_cl_half( in1.data()[idx] );
    }

    // Unmap the object -> Used in the OpenCL kernel
    CL_CHECK( clEnqueueUnmapMemObject( clQueue, clBufferA, (void*) clHostFp16PtrA,
            0, NULL, &unMapBufferEvents[0] ) );

    // Unmap the object -> Used in the OpenCL kernel
    CL_CHECK( clEnqueueUnmapMemObject( clQueue, clBufferB, (void*) clHostFp16PtrB,
            0, NULL, &unMapBufferEvents[1] ) );

    // Matrix C
    clBufferC = clCreateBuffer(clCtx, CL_MEM_HOST_READ_ONLY |
CL_MEM_ALLOC_HOST_PTR,
```

```
                c_size, NULL, NULL);

    // Wait for completion
    CL_CHECK( clWaitForEvents(2, unMapBufferEvents) );
    return CL_SUCCESS;
  }

  cl_int loadFromBinaryCompute()
  {

    unsigned char* clKernelBinaryFile = NULL;
    size_t clKernelBinSize = 0;
    // Read compiled OpenCL kernel binary file from disk
    read_file(&clKernelBinaryFile, &clKernelBinSize, "matmul.bin" );

    // Create an OpenCL program object from binary
    clProgram = CL_CHECK_ERR( clCreateProgramWithBinary(clCtx, 1, &clDevice,
                        &clKernelBinSize,
                        (const unsigned char **)&clKernelBinaryFile,
                        NULL, &_err) );

    // OpenCL build program
    CL_CHECK( clBuildProgram(clProgram, 1, &clDevice, NULL , NULL, NULL) );

    cl_ushort gemmKernelIter;
    cl_ushort transKernelIter;

    // Create OpenCL GEMM kernel object
    // clGemmKernel = CL_CHECK_ERR( clCreateKernel(clProgram,
"MatMul_TN_1D_Fp16_Half4" , &_err) );
    clGemmKernel = CL_CHECK_ERR( clCreateKernel(clProgram,
"MatMul_TN_1D_Fp16_Half8" , &_err) );
    // clGemmKernel = CL_CHECK_ERR( clCreateKernel(clProgram,
"MatMul_TN_1D_Fp16_Half16" , &_err) );

    // Create OpenCL Transpose kernel object
    // clTransKernel = CL_CHECK_ERR( clCreateKernel(clProgram,
"MatTrans_1D_Fp16_Half4" , &_err) );
    clTransKernel = CL_CHECK_ERR( clCreateKernel(clProgram,
"MatTrans_1D_Fp16_Half8" , &_err) );
    // clTransKernel = CL_CHECK_ERR( clCreateKernel(clProgram,
"MatTrans_1D_Fp16_Half16" , &_err) );

    // Handle Matrices Transpose
    if( a_traspose && b_traspose ){ // Transpose A: yes, Transpose B: yes

      transKernelIter = ColA >> 3;
      gemmKernelIter = RowA >> 3;

      // Transpose A
      SET_TRANS_KERNEL_ARG(RowA, ColA, clBufferA, clBufferA_T, transKernelIter);

      CL_CHECK( clEnqueueNDRangeKernel(clQueue, clTransKernel, 1, NULL,
            &RowA, NULL, 0, NULL, &transKernelEvent[0]) );
```

```
  SET_GEMM_TN_KERNEL_ARG(ColA, RowA, RowB, clBufferA_T, clBufferB,
    clBufferC, ColA, cl_half, gemmKernelIter);

  const size_t global = ColA;
  CL_CHECK( clEnqueueNDRangeKernel(clQueue, clGemmKernel, 1, NULL,
            &global, NULL, 1, transKernelEvent, &gemmKernelEvent) );

  CL_CHECK( clWaitForEvents(1, &gemmKernelEvent) );

}else if( a_traspose && !b_traspose ){ // Transpose A: yes, Transpose B: no

  transKernelIter = ColA >> 3;
  gemmKernelIter = RowA >> 3;

  // Transpose A
  SET_TRANS_KERNEL_ARG(RowA, ColA, clBufferA, clBufferA_T, transKernelIter);

  CL_CHECK( clEnqueueNDRangeKernel(clQueue, clTransKernel, 1, NULL,
            &RowA, NULL, 0, NULL, &transKernelEvent[0]) );

  transKernelIter = ColB >> 3;

  // Transpose B
  SET_TRANS_KERNEL_ARG(RowB, ColB, clBufferB, clBufferB_T, transKernelIter);

  CL_CHECK( clEnqueueNDRangeKernel(clQueue, clTransKernel, 1, NULL,
            &RowB, NULL, 0, NULL, &transKernelEvent[1]) );

  SET_GEMM_TN_KERNEL_ARG(ColA, RowA, ColB, clBufferA_T, clBufferB_T,
    clBufferC, RowA, cl_half, gemmKernelIter);

  const size_t global = ColA;
  CL_CHECK( clEnqueueNDRangeKernel(clQueue, clGemmKernel, 1, NULL,
            &global, NULL, 2, transKernelEvent, &gemmKernelEvent) );

  CL_CHECK( clWaitForEvents(1, &gemmKernelEvent) );

}else if( !a_traspose && b_traspose ){ // Transpose A: no, Transpose B: yes

  gemmKernelIter = ColA >> 3;

  // Transpose A
  SET_GEMM_TN_KERNEL_ARG(RowA, ColA, RowB, clBufferA, clBufferB,
    clBufferC, ColA, cl_half, gemmKernelIter);

  const size_t global = RowA;
  CL_CHECK( clEnqueueNDRangeKernel(clQueue, clGemmKernel, 1, NULL,
            &global, NULL, 0, NULL, &gemmKernelEvent) );

  CL_CHECK( clWaitForEvents(1, &gemmKernelEvent) );

}else if( !a_traspose && !b_traspose ){ // Transpose A: no, Transpose B: no
```

```
        transKernelIter = ColB >> 3;
        gemmKernelIter = ColA >> 3;

        // Transpose B
        SET_TRANS_KERNEL_ARG(ColA, ColB, clBufferB, clBufferB_T, transKernelIter);

        CL_CHECK( clEnqueueNDRangeKernel(clQueue, clTransKernel, 1, NULL,
                &ColA, NULL, 0, NULL, &transKernelEvent[0]) );

        SET_GEMM_TN_KERNEL_ARG(RowA, ColA, ColB, clBufferA, clBufferB_T,
          clBufferC, ColA, cl_half, gemmKernelIter);

        const size_t global = RowA;
        CL_CHECK( clEnqueueNDRangeKernel(clQueue, clGemmKernel, 1, NULL,
                &global, NULL, 1, transKernelEvent, &gemmKernelEvent) );

        CL_CHECK( clWaitForEvents(1, &gemmKernelEvent) );
      }
      return CL_SUCCESS;
    }

  private:
    // Copied memory data
    cl_half * clHostFp16PtrA;
    cl_half * clHostFp16PtrB;

};  // class clQualcommFP16Engine

// clBLASTEngine concrete class using CLBLAST API
class clBLASTEngine : public clMatMulEngine<float>{
  public:

    cl_int clEnd(){

      // Free OpenCL memory objects
      CL_CHECK( clReleaseMemObject(clBufferA) );
      CL_CHECK( clReleaseMemObject(clBufferB) );
      CL_CHECK( clReleaseMemObject(clBufferC) );

      // Free OpenCL command queue
      CL_CHECK( clReleaseCommandQueue(clQueue) );

      // Free OpenCL context
      CL_CHECK( clReleaseContext(clCtx) );

      // Free OpenCL events
      CL_CHECK( clReleaseEvent(gemmKernelEvent) );
      CL_CHECK( clReleaseEvent(writeBufferEvents[0]) );
      CL_CHECK( clReleaseEvent(writeBufferEvents[1]) );

      // Return CL_SUCCESS if all resources are released successfully
      return CL_SUCCESS;
    }
```

```
cl_int memLoad(typename functor::MatMulTypes<float>::out_type out){

  // Read results
  CL_CHECK( clEnqueueReadBuffer(clQueue, clBufferC, CL_TRUE, 0, c_size,
            out.data(), 0, NULL, NULL) );

  // Release OpenCL resources
  CL_CHECK( clEnd() );

  // Return if the results are loaded to memory & OpenCL resources are released
  return CL_SUCCESS;
}

cl_int memInit(
  typename functor::MatMulTypes<float>::in_type in0,
  typename functor::MatMulTypes<float>::in_type in1)
{

  // Allocate memory buffers
  clBufferA = CL_CHECK_ERR( clCreateBuffer(clCtx, CL_MEM_READ_ONLY, a_size,
                        NULL, &_err) );
  clBufferB = CL_CHECK_ERR( clCreateBuffer(clCtx, CL_MEM_READ_ONLY, b_size,
                        NULL, &_err) );
  clBufferC = CL_CHECK_ERR( clCreateBuffer(clCtx, CL_MEM_READ_WRITE, c_size,
                        NULL, &_err) );

  // Enqueue write buffer commands (acynchronous write)
  CL_CHECK( clEnqueueWriteBuffer(clQueue, clBufferA, CL_FALSE, 0, a_size,
            in0.data(), 0, NULL, &writeBufferEvents[0]) );

  CL_CHECK( clEnqueueWriteBuffer(clQueue, clBufferB, CL_FALSE, 0, b_size,
            in1.data(), 0, NULL, &writeBufferEvents[1]) );

  // Wait for completion
  CL_CHECK( clWaitForEvents(2, writeBufferEvents) );
  return CL_SUCCESS;
}

cl_int clBlastCompute()
{
  // Whether Matrix A, B should be transposed
  auto MatATranspose = ( a_traspose == true ) ?
                    CLBlastTransposeYes : CLBlastTransposeNo;
  auto MatBTranspose = ( b_traspose == true ) ?
                    CLBlastTransposeYes : CLBlastTransposeNo;

  // Leading dimension of the input A matrix. This value must be greater than 0.
  size_t a_ld;

  // Leading dimension of the input B matrix. This value must be greater than 0.
  size_t b_ld;

  // When transpose_a == Transpose::kNo, then a_ld must be at least m,
  // otherwise a_ld must be at least k.
```

```
    if( MatATranspose == CLBlastTransposeYes ){
      a_ld = RowA;
    }else{
      a_ld = ColA;
    }

    // When transpose_b == Transpose::kNo, then b_ld must be at least k,
    // otherwise b_ld must be at least n.
    if( MatBTranspose == CLBlastTransposeYes ){
      b_ld = ColA;
    }else{
      b_ld = ColB;
    }

    // The value of c_ld must be at least m.
    const size_t c_ld = ColB;

    // Performs the matrix product C = alpha * A * B + beta * C
    const float alpha = 1.0f;
    const float beta = 0.0f;

    // Call the SGEMM routine.
    CLBlastStatusCode status = CLBlastSgemm(CLBlastLayoutRowMajor,
                                    MatATranspose, MatBTranspose,
                                    RowA, ColB, ColA,
                                    alpha,
                                    clBufferA, 0, a_ld,
                                    clBufferB, 0, b_ld,
                                    beta,
                                    clBufferC, 0, c_ld,
                                    &clQueue, &gemmKernelEvent);

    // Wait for completion
    if (status != CLBlastSuccess){
      LOG(ERROR) << "[CLBlast] Fail with code " << status;
      return CL_FALSE;
    }

    CL_CHECK( clWaitForEvents(1, &gemmKernelEvent) );
    return CL_SUCCESS;
  }

 protected:

  // OpenCL memeory object
  cl_mem clBufferA;
  cl_mem clBufferB;
  cl_mem clBufferC;

  // OpenCL events
  cl_event gemmKernelEvent;
  cl_event writeBufferEvents[2];

};  // class clBLASTEngine
```

```
namespace functor {

 template <typename Device, typename T>
 struct MatMulCLFunctor {
  // Computes on device "d": out = in0 * in1, where * is matrix
  // multiplication.
  void operator()(
      const Device& d, typename MatMulTypes<T>::out_type out,
      typename MatMulTypes<T>::in_type in0,
      typename MatMulTypes<T>::in_type in1,
      const Eigen::array<Eigen::IndexPair<Eigen::DenseIndex>, 1>& dim_pair);
 };

 // Partial specialization MatMulFunctor<Device=CPUDevice, T>.
 template <typename T>
 struct MatMulCLFunctor<CPUDevice, T> {
  void operator()(
      const CPUDevice& d, typename MatMulTypes<T>::out_type out,
      typename MatMulTypes<T>::in_type in0,
      typename MatMulTypes<T>::in_type in1,
      const Eigen::array<Eigen::IndexPair<Eigen::DenseIndex>, 1>& dim_pair) {
    MatMul<CPUDevice>(d, out, in0, in1, dim_pair);
  }
 };

 // Partial specialization MatMulFunctor<Device=CPUDevice, float>
 /*
 Notice that only floating pointing matrix multiplication will be handled by
 OpenCL, other datatype complutation will be handled by Eigen CPU library
 */
 template <>
 struct MatMulCLFunctor<CPUDevice, float> {
  void operator()(
      const CPUDevice& d, typename MatMulTypes<float>::out_type out,
      typename MatMulTypes<float>::in_type in0,
      typename MatMulTypes<float>::in_type in1,
      const Eigen::array<Eigen::IndexPair<Eigen::DenseIndex>, 1>& dim_pair)
    {

    clQualcommFP32Engine c = clQualcommFP32Engine();
    // clQualcommFP16Engine c = clQualcommFP16Engine();
    // clBLASTEngine c = clBLASTEngine();

    // OpenCL host & device side initializaiotn
    CL_CHECK( c.hostInit(in0, in1, out, dim_pair) );

    // debug info
    // c.debug(true);

    // OpenCL memeory object init & memory copy
    CL_CHECK( c.memInit(in0, in1) );

    // GEMM computation
```

```
      CL_CHECK( c.loadFromBinaryCompute() );
      // CL_CHECK( c.clBlastCompute() );

      // OpenCL memory load
      CL_CHECK( c.memLoad(out) );

      // Results
      // c.printMatrix(in0, in1, out);

    }
  };

}  // end namespace functor
}  // end namespace tensorflow

#endif  // MATMUL_CL_FUNCTOR_H_
```

B. MNIST AI model building Python script

```
# Copyright 2015 The TensorFlow Authors. All Rights Reserved.
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#     http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.
#
==============================================================================
========

"""A very simple MNIST classifier.

See extensive documentation at
https://www.tensorflow.org/get_started/mnist/beginners
"""
from __future__ import absolute_import
from __future__ import division
from __future__ import print_function

import argparse
import sys
import tensorflow as tf
from tensorflow.examples.tutorials.mnist import input_data

def main(_):

    # Import data
    mnist = input_data.read_data_sets(FLAGS.mnistDataDir, one_hot=True)

    # Training parameters
    maxEpochs = FLAGS.maxEpochs
    batchSize = FLAGS.batchSize
    testStep = FLAGS.testStep

    # Network parameters
    n_hidden_1 = 50 # 1st layer number of neurons
    n_hidden_2 = 50 # 2nd layer number of neurons
    n_input = 784 # MNIST data input (img shape: 28*28)
    n_classes = 10 # MNIST total classes (0-9 digits)

    # tf Graph input
    X = tf.placeholder(tf.float32, [None, n_input], name="input")
    Y = tf.placeholder(tf.float32, [None, n_classes], name="output")

    # Store layers weight & bias
    weights = {
        'h1': tf.Variable(tf.random_normal([n_input, n_hidden_1])),
```

```
    'h2': tf.Variable(tf.random_normal([n_hidden_1, n_hidden_2])),
    'out': tf.Variable(tf.random_normal([n_hidden_2, n_classes]))
}
biases = {
    'b1': tf.Variable(tf.random_normal([n_hidden_1])),
    'b2': tf.Variable(tf.random_normal([n_hidden_2])),
    'out': tf.Variable(tf.random_normal([n_classes]))
}

# Create model
def multilayer_perceptron(x):
    # Hidden fully connected layer with `n_hidden_1` neurons
    layer_1 = tf.add(tf.matmul(x, weights['h1']), biases['b1'])
    # Hidden fully connected layer with `n_hidden_2` neurons
    layer_2 = tf.add(tf.matmul(layer_1, weights['h2']), biases['b2'])
    # Output fully connected layer with a neuron for each class
    out_layer = tf.matmul(layer_2, weights['out']) + biases['out']
    return out_layer

# Construct model
logits = multilayer_perceptron(X)

# Define loss
loss = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits_v2(
    logits=logits, labels=Y))

# Define optimizer
with tf.name_scope('adam_optimizer'):
    train_op = tf.train.AdamOptimizer().minimize(loss, name="train")

# Define accuracy
prediction = tf.equal(tf.argmax(logits, 1), tf.argmax(Y, 1))
accuracy = tf.reduce_mean(tf.cast(prediction, tf.float32), name="test")

# Create a summary to monitor cross_entropy tensor
tf.summary.scalar("loss", loss)
# Create a summary to monitor accuracy tensor
tf.summary.scalar("accuracy", accuracy)
# Merge all summaries into a single op
merged_summary_op = tf.summary.merge_all()

# Initializing the variables
init = tf.initialize_variables(tf.all_variables(), name='init')

with tf.Session() as sess:
    # Session Init
    sess.run(init)

    # Logger Init
    summaryWriter = tf.summary.FileWriter(FLAGS.logDir, graph=sess.graph)

    # Training
    for step in range(maxEpochs):
        # Get MNIST training data
```

```
            batchImage, batchLabel = mnist.train.next_batch(batchSize)

            # Test training model for every testStep
            if step % testStep == 0:
                # Run accuracy op & summary op to get accuracy & training progress
                acc, summary = sess.run([accuracy, merged_summary_op],
                    feed_dict={X: mnist.test.images, Y: mnist.test.labels})

                # Write accuracy to log file
                summaryWriter.add_summary(summary, step)

                # Print accuracy
                print('step %d, training accuracy %f' % (step, acc))

            # Run training op
            train_op.run( feed_dict={ X: batchImage, Y: batchLabel })

        # Write TF model
        tf.train.write_graph(sess.graph_def,
                             './',
                             'mnist_mlp.pb', as_text=False)

if __name__ == '__main__':
    parser = argparse.ArgumentParser()
    parser.add_argument('--mnistDataDir', type=str, default='/tmp/tensorflow/mnist/input_data',
                        help='MNIST data directory')
    parser.add_argument('--logDir', type=str, default='/tmp/tensorflow_logs/mlpnet',
                        help='Training progress data directory')
    parser.add_argument('--batchSize', type=int, default=50,
                        help='Training batch size')
    parser.add_argument('--maxEpochs', type=int, default=10000,
                        help='Maximum training steps')
    parser.add_argument('--testStep', type=int, default=100,
                        help='Test model accuracy for every testStep iterations')
    FLAGS, unparsed = parser.parse_known_args()
    # Program entry
    tf.app.run(main=main, argv=[sys.argv[0]] + unparsed)
```

```
# Copyright 2015 The TensorFlow Authors. All Rights Reserved.
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#     http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.
#
==============================================================================
========

"""A deep MNIST classifier using convolutional layers.

See extensive documentation at
https://www.tensorflow.org/get_started/mnist/pros
"""
from __future__ import absolute_import
from __future__ import division
from __future__ import print_function

import argparse
import sys
import tensorflow as tf
from tensorflow.examples.tutorials.mnist import input_data

def deepnn(X):
    """deepnn builds the graph for a deep net for classifying digits.

    Args:
    X: an input tensor with the dimensions (N_examples, 784), where 784 is the
    number of pixels in a standard MNIST image.

    Returns:
    A tuple (y, keepProb). y is a tensor of shape (N_examples, 10), with values
    equal to the logits of classifying the digit into one of 10 classes (the
    digits 0-9). keepProb is a scalar placeholder for the probability of
    dropout.
    """
    # Reshape to use within a convolutional neural net.
    # Last dimension is for "features" - there is only one here, since images are
    # grayscale -- it would be 3 for an RGB image, 4 for RGBA, etc.
    with tf.name_scope('reshape'):
        x_image = tf.reshape(X, [-1, 28, 28, 1])

    # First convolutional layer - maps one grayscale image to 32 feature maps.
    with tf.name_scope('conv1'):
        W_conv1 = weight_variable([5, 5, 1, 32])
        b_conv1 = bias_variable([32])
```

```
        h_conv1 = tf.nn.relu(conv2d(x_image, W_conv1) + b_conv1)

    # Pooling layer - downsamples by 2X.
    with tf.name_scope('pool1'):
        h_pool1 = max_pool_2x2(h_conv1)

    # Second convolutional layer -- maps 32 feature maps to 64.
    with tf.name_scope('conv2'):
        W_conv2 = weight_variable([5, 5, 32, 64])
        b_conv2 = bias_variable([64])
        h_conv2 = tf.nn.relu(conv2d(h_pool1, W_conv2) + b_conv2)

    # Second pooling layer.
    with tf.name_scope('pool2'):
        h_pool2 = max_pool_2x2(h_conv2)

    # Fully connected layer 1 -- after 2 round of downsampling, our 28x28 image
    # is down to 7x7x64 feature maps -- maps this to 1024 features.
    with tf.name_scope('fc1'):
        W_fc1 = weight_variable([7 * 7 * 64, 1024])
        b_fc1 = bias_variable([1024])
        h_pool2_flat = tf.reshape(h_pool2, [-1, 7 * 7 * 64])
        h_fc1 = tf.nn.relu(tf.matmul(h_pool2_flat, W_fc1) + b_fc1)

    # Dropout - controls the complexity of the model, prevents co-adaptation of
    # features.
    with tf.name_scope('Dropout'):
        keepProb = tf.placeholder(tf.float32)
        h_fc1_drop = tf.nn.dropout(h_fc1, keepProb)

    # Map the 1024 features to 10 classes, one for each digit
    with tf.name_scope('fc2'):
        W_fc2 = weight_variable([1024, 10])
        b_fc2 = bias_variable([10])
        Yconv = tf.matmul(h_fc1_drop, W_fc2) + b_fc2

    return Yconv, keepProb

def conv2d(X, W):
    """conv2d returns a 2d convolution layer with full stride."""
    return tf.nn.conv2d(X, W, strides=[1, 1, 1, 1], padding='SAME')


def max_pool_2x2(X):
    """max_pool_2x2 downsamples a feature map by 2X."""
    return tf.nn.max_pool(X, ksize=[1, 2, 2, 1],
                          strides=[1, 2, 2, 1], padding='SAME')


def weight_variable(shape):
    """weight_variable generates a weight variable of a given shape."""
    initial = tf.truncated_normal(shape, stddev=0.1)
    return tf.Variable(initial)
```

```
def bias_variable(shape):
    """bias_variable generates a bias variable of a given shape."""
    initial = tf.constant(0.1, shape=shape)
    return tf.Variable(initial)


def main(_):

    # Import data
    mnist = input_data.read_data_sets(FLAGS.mnistDataDir, one_hot=True)

    # Training parameters
    maxEpochs = FLAGS.maxEpochs
    batchSize = FLAGS.batchSize
    testStep = FLAGS.testStep

    # Network parameters
    n_input = 784 # MNIST data input (img shape: 28*28)
    n_classes = 10 # MNIST total classes (0-9 digits)

    # Create the model
    X = tf.placeholder(tf.float32, [None, n_input], name="input")
    Y = tf.placeholder(tf.float32, [None, n_classes], name="output")

    # Build the graph for the deep net
    Yconv, keepProb = deepnn(X)

    # Define loss
    loss = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits_v2(
        logits=Yconv, labels=Y))

    # Define optimizer
    with tf.name_scope('adam_optimizer'):
        train_op = tf.train.AdamOptimizer().minimize(loss, name="train")

    # Define accuracy
    prediction = tf.equal(tf.argmax(Yconv, 1), tf.argmax(Y, 1))
    accuracy = tf.reduce_mean(tf.cast(prediction, tf.float32), name="test")

    # Create a summary to monitor cross_entropy tensor
    tf.summary.scalar("loss", loss)
    # Create a summary to monitor accuracy tensor
    tf.summary.scalar("accuracy", accuracy)
    # Merge all summaries into a single op
    merged_summary_op = tf.summary.merge_all()

    # Initializing the variables
    init = tf.initialize_variables(tf.all_variables(), name='init')

    with tf.Session() as sess:
        # Session Init
        sess.run(init)
```

```
        # Logger Init
        summaryWriter = tf.summary.FileWriter(FLAGS.logDir, graph=sess.graph)

        # Training
        for step in range( maxEpochs ):
            # Get MNIST training data
            batchImage, batchLabel = mnist.train.next_batch(batchSize)

            # Test training model for every testStep
            if step % testStep == 0:
                # Run accuracy op & summary op to get accuracy & training progress
                acc, summary = sess.run( [ accuracy, merged_summary_op ], \
                    feed_dict={ X: mnist.test.images, Y: mnist.test.labels, keepProb: 1.0})

                # Write accuracy to log file
                summaryWriter.add_summary(summary, step)

                # Print accuracy
                print('step %d, training accuracy %f' % (step, acc))

            # Run training op
            train_op.run(feed_dict={X: batchImage, Y: batchLabel, keepProb: 0.5})

        # Write TF model
        tf.train.write_graph(sess.graph_def,
                             './',
                             'mnist_dnn.pb', as_text=False)

if __name__ == '__main__':
    parser = argparse.ArgumentParser()
    parser.add_argument('--mnistDataDir', type=str, default='/tmp/tensorflow/mnist/input_data',
                        help='MNIST data directory')
    parser.add_argument('--logDir', type=str, default='/tmp/tensorflow_logs/deepnet',
                        help='Training progress data directory')
    parser.add_argument('--batchSize', type=int, default=50,
                        help='Training batch size')
    parser.add_argument('--maxEpochs', type=int, default=10000,
                        help='Maximum training steps')
    parser.add_argument('--testStep', type=int, default=100,
                        help='Test model accuracy for every testStep iterations')
    FLAGS, unparsed = parser.parse_known_args()
    # Program entry
    tf.app.run(main=main, argv=[sys.argv[0]] + unparsed)
```

```
# Description:
#   TensorFlow C++ training example for MNIST dataset

package(
    default_visibility = ["//tensorflow:internal"],
)

licenses(["notice"])  # Apache 2.0

exports_files(["LICENSE"])

load(
    "//tensorflow:tensorflow.bzl",
    "tf_copts",
)

ANDROID_C_OPTS = tf_copts() + [
    "-ffunction-sections",
    "-fdata-sections",
    "-fPIE",
    "-pie",
    "-fexceptions",
]

ANDROID_LINK_OPTS = [
    "-fPIE",
    "-pie",
    "-landroid",
    "-latomic",
    "-ldl",
    "-llog",
    "-lm",
    "-z defs",
    "-s",
    "-Wl,--gc-sections",
    "-fuse-ld=gold",
]

cc_library(
    name = "mnistReader",
    srcs = [
        "mnistReader.cc",
    ],
    hdrs = [
        "mnistReader.h",
    ],
    deps = [
        "//external:libmnist",
        "//tensorflow/core:android_tensorflow_lib",
    ],
)

cc_library(
    name = "tf_runner_lib",
```

```
    srcs = [
        "tfRunner.cc",
    ],
    hdrs = [
        "tfRunner.h",
    ],
    deps = [
        "//tensorflow/cc:cc_ops",
        "//tensorflow/core:android_tensorflow_lib",
    ],
)

cc_binary(
  name = "train_and_test_mnist",
  srcs = [
      "train_and_test.cc",
      "util.h",
      "tfRunner.h",
      "tfRunner.cc",
  ],
  copts = ANDROID_C_OPTS,
  linkopts = ANDROID_LINK_OPTS,
  deps = [
      "//tensorflow/cc:cc_ops",
      "//tensorflow/core:android_tensorflow_lib",
      ":mnistReader",
      ":tf_runner_lib",
      "//external:libTFlogger",
  ],
)


cc_binary(
  name = "train_mnist",
  srcs = [
      "train_mnist.cc",
      "util.h",
      "tfRunner.h",
      "tfRunner.cc",
  ],
  copts = ANDROID_C_OPTS,
  linkopts = ANDROID_LINK_OPTS,
  deps = [
      "//tensorflow/cc:cc_ops",
      "//tensorflow/core:android_tensorflow_lib",
      ":mnistReader",
      ":tf_runner_lib",
  ],
)
```

```
/*
By Cheng Wei on 2018/Jan/24
================================================================================*/
// A simple program trainging a MNIST TF model using TF C++ API

#include <vector>
#include <chrono>

#include "tensorflow/core/framework/graph.pb.h"
#include "tensorflow/core/framework/tensor.h"
#include "tensorflow/core/graph/default_device.h"
#include "tensorflow/core/graph/graph_def_builder.h"
#include "tensorflow/core/lib/core/errors.h"
#include "tensorflow/core/lib/core/stringpiece.h"
#include "tensorflow/core/lib/core/threadpool.h"
#include "tensorflow/core/lib/io/path.h"
#include "tensorflow/core/lib/strings/stringprintf.h"
#include "tensorflow/core/platform/env.h"
#include "tensorflow/core/platform/init_main.h"
#include "tensorflow/core/platform/logging.h"
#include "tensorflow/core/platform/types.h"
#include "tensorflow/core/public/session.h"
#include "tensorflow/core/util/command_line_flags.h"

#include "tfRunner.h"
#include "util.h"
#include "mnistReader.h"

// These are all common classes it's handy to reference with no namespace.
using tensorflow::Flag;
using tensorflow::Tensor;
using tensorflow::Status;
using tensorflow::string;
using tensorflow::int32;
using namespace tensorflow;
using namespace std;

int main(int argc, char* argv[]) {

  string root_dir         = "/data/local/tmp/";
  string graphName        = "mnist_mlp.pb";
  string mnistDir         = root_dir + "MNIST_data/";
  string inputOpsName     = "input";
  string outputOpsName    = "output";
  string accuOpsName      = "test";
  string trainOpsName     = "adam_optimizer/train";
  string dropoutOpsName   = "Dropout/Placeholder";
  int32  input_width      = 28;
  int32  input_height     = 28;
  int32  batchSize        = 50;
  int32  maxSteps         = 1000000;
  float  iteration        = 1.0f;
```

```
vector<float> dropProb  = { 0.5 } ;

// Start the timer
auto start_time = std::chrono::high_resolution_clock::now();

vector<Flag> flag_list = {
    Flag("root_dir",      &root_dir,      "Binary Root Directory"),
    Flag("graphName",     &graphName,     "Graph To Be Executed"),
    Flag("mnistDir",      &mnistDir,      "MNIST Dataset Directory"),
    Flag("inputOpsName",  &inputOpsName,  "Input Ops Name"),
    Flag("outputOpsName", &outputOpsName, "Output Ops Name"),
    Flag("accuOpsName",   &accuOpsName,   "Cost Ops Name"),
    Flag("trainOpsName",  &trainOpsName,  "Train Ops Name"),
    Flag("dropoutOpsName",&dropoutOpsName,"Dropout Ops Name"),
    Flag("batchSize",     &batchSize,     "Training & Testing Batch Size"),
    Flag("maxSteps",      &maxSteps,      "Maximum Number of Taining Steps"),
    Flag("iteration",     &iteration,     "Number of Iteration to Traing the Whole Dataset"),
    Flag("dropProb",      &dropProb[0],   "Drop-out Layer (if any) Probability"),
};

string usage = Flags::Usage(argv[0], flag_list);
const bool parse_result = Flags::Parse(&argc, argv, flag_list);
if (!parse_result) {
  LOG(ERROR) << usage;
  return -1;
}

// We need to call this to set up global state for TensorFlow.
port::InitMain(argv[0], &argc, &argv);
if (argc > 1) {
  LOG(ERROR) << "Unknown argument " << argv[1] << "\n" << usage;
  return -1;
}

LOG(INFO) << "[Root directory] = " << root_dir ;

// Prepare MNIST dataset
LOG(INFO) << "[MNIST Dataset Directory] = " << mnistDir ;

mnistReader mnist = mnistReader(mnistDir);

LOG(INFO) << "[MNIST Dataset] Num of Training Images = " << mnist.getTrainingDataSize();
LOG(INFO) << "[MNIST Dataset] Num of Training Labels = " << mnist.getTrainingDataSize();
LOG(INFO) << "[MNIST Dataset] Num of Test     Images = " << mnist.getTestingDataSize();
LOG(INFO) << "[MNIST Dataset] Num of Test     Labels = " << mnist.getTestingDataSize();
LOG(INFO) << "[MNIST Dataset] Input Image Size       = " << mnist.getImgSize();

input_width  = mnist.getImgSize();
input_height = mnist.getImgSize();

// Load TF model.
unique_ptr<Session> session;
string graph_path = io::JoinPath(root_dir, graphName);
Status load_graph_status = LoadGraph(graph_path, &session);
```

```
  if (!load_graph_status.ok()) {
    LOG(ERROR) << load_graph_status;
    return -1;
  }
  LOG(INFO) << "[TF Model File Loaded From Directory] = " << graph_path ;

  tfRunner runner = tfRunner("init", trainOpsName, accuOpsName, graphName);

  runner.sessInit(session);

  runner.tensorInit(batchSize, input_width*input_height);

  for( auto beginIdx = 0 ; beginIdx < mnist.getTrainingDataSize()*iteration - batchSize;
    beginIdx = beginIdx + batchSize )
  {

    LOG(INFO) << beginIdx << " trained.";

    // If the number of training steps > maxSteps then stop training
    if ( beginIdx > maxSteps ){ break; }

    // image vector with dimension { 1, batchSize x input_width x input_height }
    vector<float> batchTrainImgFloatVec;
    // label vector with dimension { 1, batchSize }
    vector<float> batchTrainLabelFloatVec;

    mnist.getTrainingBatch(beginIdx, batchSize, &batchTrainImgFloatVec,
&batchTrainLabelFloatVec);

    runner.copyToTensor(batchTrainImgFloatVec, batchTrainLabelFloatVec, dropProb);

    runner.sessionTrain(session, inputOpsName, outputOpsName, dropoutOpsName);

  } // End of Training Batch Loop

  auto elapsed_time = std::chrono::high_resolution_clock::now() - start_time;
  auto time_s = std::chrono::duration<double>(elapsed_time).count();

  LOG(INFO) << "Training " << graphName << " takes " << time_s << " sec";

  vector<double> avg_accu;

  batchSize = 100;

  for( auto beginIdx = 0 ; beginIdx < mnist.getTestingDataSize() - batchSize;
    beginIdx = beginIdx + batchSize )
  { // Testing Batch Loop

    LOG(INFO) << beginIdx << " tested.";

    // image vector with dimension { 1, batchSize x input_width x input_height }
    vector<float> batchTestImgFloatVec;
    // label vector with dimension { 1, batchSize }
    vector<float> batchTestLabelFloatVec;
```

```
    mnist.getTestingBatch(beginIdx, batchSize, &batchTestImgFloatVec,
&batchTestLabelFloatVec);

    // No drop out layer when testing
    dropProb[0] = 1.0f;

    runner.copyToTensor(batchTestImgFloatVec, batchTestLabelFloatVec, dropProb);

    double acc = runner.sessionTest(session, inputOpsName, outputOpsName,
dropoutOpsName);

    avg_accu.push_back( acc );
    LOG(INFO) << "Accuracy " << acc * 100 << "\%";

  } // End of Testing Batch Loop

  auto acc = 100 * accumulate( avg_accu.begin(), avg_accu.end(), 0.0f) / avg_accu.size();

  LOG(INFO) << "Overall testing accuracy " << acc << "\%";

} // End of main
```

D. MNIST training logger program source code

```
/*
By Cheng Wei on 2018/Jan/24
================================================================================*/
// A simple program trainging a MNIST TF model using TF C++ API

#include <vector>
#include <chrono>

#include "tensorflow/core/framework/graph.pb.h"
#include "tensorflow/core/framework/tensor.h"
#include "tensorflow/core/graph/default_device.h"
#include "tensorflow/core/graph/graph_def_builder.h"
#include "tensorflow/core/lib/core/errors.h"
#include "tensorflow/core/lib/core/stringpiece.h"
#include "tensorflow/core/lib/core/threadpool.h"
#include "tensorflow/core/lib/io/path.h"
#include "tensorflow/core/lib/strings/stringprintf.h"
#include "tensorflow/core/platform/env.h"
#include "tensorflow/core/platform/init_main.h"
#include "tensorflow/core/platform/logging.h"
#include "tensorflow/core/platform/types.h"
#include "tensorflow/core/public/session.h"
#include "tensorflow/core/util/command_line_flags.h"

#include "tfRunner.h"
#include "util.h"
#include "mnistReader.h"
#include "tensorboard_logger.h"

// These are all common classes it's handy to reference with no namespace.
using tensorflow::Flag;
using tensorflow::Tensor;
using tensorflow::Status;
using tensorflow::string;
using tensorflow::int32;
using namespace tensorflow;
using namespace std;

int main(int argc, char* argv[]) {

  string root_dir        = "/data/local/tmp/";
  string graphName       = "mnist_mlp.pb";
  string mnistDir        = root_dir + "MNIST_data/";
  string inputOpsName    = "input";
  string outputOpsName   = "output";
  string accuOpsName     = "test";
  string trainOpsName    = "adam_optimizer/train";
  string dropoutOpsName  = "Dropout/Placeholder";
  int32  input_width     = 28;
  int32  input_height    = 28;
  int32  batchSize       = 50;
  int32  maxSteps        = 1000000;
```

```
float  iteration        = 1.0f;
vector<float> dropProb  = { 0.5 } ;

int timeStamp = std::chrono::duration_cast<std::chrono::milliseconds>
( std::chrono::system_clock::now().time_since_epoch() ).count();
string logFileName      = root_dir + "events.out.tfevents." + to_string(timeStamp)
  + ".wei.local";

vector<Flag> flag_list = {
   Flag("root_dir",      &root_dir,      "Binary Root Directory"),
   Flag("graphName",     &graphName,     "Graph To Be Executed"),
   Flag("mnistDir",      &mnistDir,      "MNIST Dataset Directory"),
   Flag("inputOpsName",  &inputOpsName,  "Input Ops Name"),
   Flag("outputOpsName", &outputOpsName, "Output Ops Name"),
   Flag("accuOpsName",   &accuOpsName,   "Cost Ops Name"),
   Flag("trainOpsName",  &trainOpsName,  "Train Ops Name"),
   Flag("dropoutOpsName",&dropoutOpsName,"Dropout Ops Name"),
   Flag("batchSize",     &batchSize,     "Training & Testing Batch Size"),
   Flag("maxSteps",      &maxSteps,      "Maximum Number of Taining Steps"),
   Flag("iteration",     &iteration,     "Number of Iteration to Traing the Whole Dataset"),
   Flag("dropProb",      &dropProb[0],   "Drop-out Layer (if any) Probability"),
};

string usage = Flags::Usage(argv[0], flag_list);
const bool parse_result = Flags::Parse(&argc, argv, flag_list);
if (!parse_result) {
  LOG(ERROR) << usage;
  return -1;
}

// We need to call this to set up global state for TensorFlow.
port::InitMain(argv[0], &argc, &argv);
if (argc > 1) {
  LOG(ERROR) << "Unknown argument " << argv[1] << "\n" << usage;
  return -1;
}

LOG(INFO) << "[Root directory] = " << root_dir ;

// Prepare MNIST dataset
LOG(INFO) << "[MNIST Dataset Directory] = " << mnistDir ;

mnistReader mnist = mnistReader(mnistDir);

LOG(INFO) << "[MNIST Dataset] Num of Training Images = " << mnist.getTrainingDataSize();
LOG(INFO) << "[MNIST Dataset] Num of Training Labels = " << mnist.getTrainingDataSize();
LOG(INFO) << "[MNIST Dataset] Num of Test     Images = " << mnist.getTestingDataSize();
LOG(INFO) << "[MNIST Dataset] Num of Test     Labels = " << mnist.getTestingDataSize();
LOG(INFO) << "[MNIST Dataset] Input Image Size       = " << mnist.getImgSize();

input_width  = mnist.getImgSize();
input_height = mnist.getImgSize();

TensorBoardLogger logger( logFileName.c_str() );
```

```
  // Load TF model.
  unique_ptr<Session> session;
  string graph_path = io::JoinPath(root_dir, graphName);
  Status load_graph_status = LoadGraph(graph_path, &session);
  if (!load_graph_status.ok()) {
    LOG(ERROR) << load_graph_status;
    return -1;
  }
  LOG(INFO) << "[TF Model File Loaded From Directory] = " << graph_path ;

  tfRunner runner = tfRunner("init", trainOpsName, accuOpsName, graphName);

  runner.sessInit(session);

  runner.tensorInit(batchSize, input_width*input_height);

  for( auto beginIdx = 0 ; beginIdx < mnist.getTrainingDataSize()*iteration - batchSize;
    beginIdx = beginIdx + batchSize )
  {

    LOG(INFO) << beginIdx << " trained.";

    // If the number of training steps > maxSteps then stop training
    if ( beginIdx > maxSteps ){ break; }

    // image vector with dimension { 1, batchSize x input_width x input_height }
    vector<float> batchTrainImgFloatVec;
    // label vector with dimension { 1, batchSize }
    vector<float> batchTrainLabelFloatVec;

    mnist.getTrainingBatch(beginIdx, batchSize, &batchTrainImgFloatVec,
&batchTrainLabelFloatVec);

    runner.copyToTensor(batchTrainImgFloatVec, batchTrainLabelFloatVec, dropProb);

    runner.sessionTrain(session, inputOpsName, outputOpsName, dropoutOpsName);

    // Do overall testing for each 1000 data trained
    if( beginIdx % (5*batchSize) == 0 )
    {
      vector<double> avg_accu;

      for( auto beginIdx = 0 ; beginIdx < mnist.getTestingDataSize() - batchSize;
        beginIdx = beginIdx + batchSize )
      { // Testing Batch Loop

        // LOG(INFO) << beginIdx << " tested.";

        // image vector with dimension { 1, batchSize x input_width x input_height }
        vector<float> batchTestImgFloatVec;
        // label vector with dimension { 1, batchSize }
        vector<float> batchTestLabelFloatVec;
```

```
        mnist.getTestingBatch(beginIdx, batchSize, &batchTestImgFloatVec,
&batchTestLabelFloatVec);

        // No drop out layer when testing
        dropProb[0] = 1.0f;

        runner.copyToTensor(batchTestImgFloatVec, batchTestLabelFloatVec, dropProb);

        double acc = runner.sessionTest(session, inputOpsName, outputOpsName,
dropoutOpsName);

        avg_accu.push_back( acc );
        // LOG(INFO) << "Accuracy " << acc * 100 << "\%";

      } // End of Testing Batch Loop

      auto acc = 100 * accumulate( avg_accu.begin(), avg_accu.end(), 0.0f) / avg_accu.size();

      LOG(INFO) << "Overall testing accuracy " << acc << "\%";

      logger.add_scalar("accurarcy", beginIdx, acc);
    }

  } // End of Training Batch Loop

} // End of main
```

```
#include "CL/cl.h"

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <iostream>

using namespace std;

///
//  Attempt to create the program object from a cached binary.
///
cl_program CreateProgramFromBinary(cl_context context, cl_device_id
device, const char* fileName)
{
    FILE *fp = fopen(fileName, "rb");
    if (fp == NULL)
    {
        return NULL;
    }

    // Determine the size of the binary
    size_t binarySize;
    fseek(fp, 0, SEEK_END);
    binarySize = ftell(fp);
    rewind(fp);

    unsigned char *programBinary = new unsigned char[binarySize];
    fread(programBinary, 1, binarySize, fp);
    fclose(fp);

    cl_int errNum = 0;
    cl_program program;
    cl_int binaryStatus;

    program = clCreateProgramWithBinary(context,
                                        1,
                                        &device,
                                        &binarySize,
                                        (const unsigned
char**)&programBinary,
                                        &binaryStatus,
                                        &errNum);
    delete [] programBinary;
    if (errNum != CL_SUCCESS)
    {
        std::cerr << "Error loading program binary." << std::endl;
        return NULL;
    }

    if (binaryStatus != CL_SUCCESS)
    {
        std::cerr << "Invalid binary for device" << std::endl;
        return NULL;
```

```
    }

    errNum = clBuildProgram(program, 0, NULL, NULL, NULL, NULL);
    if (errNum != CL_SUCCESS)
    {
        // Determine the reason for the error
        char buildLog[16384];
        clGetProgramBuildInfo(program, device, CL_PROGRAM_BUILD_LOG,
                              sizeof(buildLog), buildLog, NULL);

        std::cerr << "Error in program: " << std::endl;
        std::cerr << buildLog << std::endl;
        clReleaseProgram(program);
        return NULL;
    }

    return program;
}

///
// Read in binary files
///
int read_file(char **output, size_t *size, const char *name) {
  FILE *fp = fopen(name, "rb");
  if (!fp) {
    return -1;
  }

  fseek(fp, 0, SEEK_END);
  *size = ftell(fp);
  fseek(fp, 0, SEEK_SET);

  *output = (char *)malloc(*size);
  if (!*output) {
    fclose(fp);
    return -1;
  }

  fread(*output, *size, 1, fp);
  fclose(fp);
  return 0;
}

///
// Write compiled files
///
int write_file(const char *name, const unsigned char *content,
size_t size) {
  FILE *fp = fopen(name, "wb+");
  if (!fp) {
    return -1;
  }
  fwrite(content, size, 1, fp);
  fclose(fp);
```

```
  return 0;
}

// OpenCL helper functions
cl_int get_platform_list(cl_platform_id **platforms_out,
                         cl_uint *num_platforms_out) {
  cl_int err;

  // Read the number of platforms
  cl_uint num_platforms;
  err = clGetPlatformIDs(0, NULL, &num_platforms);
  if (err != CL_SUCCESS) {
    return err;
  }
  if (num_platforms == 0) {
    return CL_INVALID_VALUE;
  }

  // Allocate the array of cl_platform_id
  cl_platform_id *platforms =
    (cl_platform_id *)malloc(sizeof(cl_platform_id) *
num_platforms);
  if (!platforms) {
    return CL_OUT_OF_HOST_MEMORY;
  }

  // Get the result
  err = clGetPlatformIDs(num_platforms, platforms, NULL);
  if (err != CL_SUCCESS) {
    free(platforms);
    return err;
  }

  *platforms_out = platforms;
  *num_platforms_out = num_platforms;
  return CL_SUCCESS;
}

void free_platform_list(cl_platform_id *platforms, cl_uint
num_platforms) {
  free(platforms);
}

char *get_platform_info(cl_platform_id platform, cl_platform_info
param) {
  cl_int err;

  // Read the size of the buffer for platform name
  size_t buf_size;
  err = clGetPlatformInfo(platform, param, 0, NULL, &buf_size);
  if (err != CL_SUCCESS) {
    return NULL;
  }
  if (buf_size == 0) {
```

```
    return NULL;
  }

  // Allocate the buffer for platform name
  char *buf = (char *)malloc(buf_size);
  if (!buf) {
    return NULL;
  }

  // Read the platform name
  err = clGetPlatformInfo(platform, param, buf_size, buf, NULL);
  if (err != CL_SUCCESS) {
    free(buf);
    return NULL;
  }

  return buf;
}

cl_int get_device_list(cl_device_id **devices_out, cl_uint
*num_devices_out,
                       cl_platform_id platform) {
  cl_int err;

  // Read the number of devices of the given platform
  cl_uint num_devices;
  err = clGetDeviceIDs(platform, CL_DEVICE_TYPE_ALL, 0, NULL,
                       &num_devices);
  if (err != CL_SUCCESS) {
    return err;
  }

  // Allocate the array of cl_device_id
  cl_device_id *devices =
    (cl_device_id *)malloc(sizeof(cl_device_id) * num_devices);
  if (!devices) {
    return CL_OUT_OF_HOST_MEMORY;
  }

  // Read the result
  err = clGetDeviceIDs(platform, CL_DEVICE_TYPE_ALL, num_devices,
                       devices, NULL);
  if (err != CL_SUCCESS) {
    free(devices);
    return err;
  }

  *devices_out = devices;
  *num_devices_out = num_devices;
  return CL_SUCCESS;
}

void free_device_list(cl_device_id *devices, cl_uint num_devices) {
  cl_uint i;
```

```
  for (i = 0; i < num_devices; ++i) {
    clReleaseDevice(devices[i]);
  }
  free(devices);
}

cl_int write_binaries(cl_program program, unsigned num_devices,
                      cl_uint platform_idx, const char *
outputBinaryName ) {
  unsigned i;
  cl_int err = CL_SUCCESS;
  size_t *binaries_size = NULL;
  unsigned char **binaries_ptr = NULL;

  // Read the binaries size
  size_t binaries_size_alloc_size = sizeof(size_t) * num_devices;
  binaries_size = (size_t *)malloc(binaries_size_alloc_size);
  if (!binaries_size) {
    err = CL_OUT_OF_HOST_MEMORY;
    return err;
  }

  err = clGetProgramInfo(program, CL_PROGRAM_BINARY_SIZES,
                         binaries_size_alloc_size, binaries_size,
NULL);
  if (err != CL_SUCCESS) {
    return err;
  }

  // Read the binaries
  size_t binaries_ptr_alloc_size = sizeof(unsigned char *) *
num_devices;
  binaries_ptr = (unsigned char **)malloc(binaries_ptr_alloc_size);
  if (!binaries_ptr) {
    err = CL_OUT_OF_HOST_MEMORY;
    return err;
  }
  memset(binaries_ptr, 0, binaries_ptr_alloc_size);
  for (i = 0; i < num_devices; ++i) {
    binaries_ptr[i] = (unsigned char *)malloc(binaries_size[i]);
    if (!binaries_ptr[i]) {
      err = CL_OUT_OF_HOST_MEMORY;
      return err;
    }
  }

  err = clGetProgramInfo(program, CL_PROGRAM_BINARIES,
binaries_ptr_alloc_size,
                         binaries_ptr, NULL);
  if (err != CL_SUCCESS) {
    return err;
  }

  // Write the binaries to file
```

```
  for (i = 0; i < num_devices; ++i) {
    // Write the binary to the output file
    write_file(outputBinaryName, binaries_ptr[i], binaries_size[i]);
  }

  return err;
}

cl_int compile_program(cl_uint *num_devices_out, const char *src,
                       size_t src_size, cl_platform_id platform,
                       cl_uint platform_idx, const char *
outputBinaryName ) {
  cl_int err = CL_SUCCESS;

  // Get the device list
  cl_device_id* devices = NULL;
  cl_uint num_devices = 0;
  get_device_list(&devices, &num_devices, platform);
  *num_devices_out = num_devices;

  // Create context
  cl_context_properties ctx_properties[] = {
    CL_CONTEXT_PLATFORM, (cl_context_properties)platform, 0
  };

  cl_context ctx = clCreateContext(ctx_properties, num_devices,
devices, NULL,
                                   NULL, &err);
  if (err != CL_SUCCESS) {
    return err;
  }

  // Create program
  cl_program program = clCreateProgramWithSource(ctx, 1, &src,
&src_size, &err);
  if (err != CL_SUCCESS) {
    return err;
  }

  // Compile program
  err = clBuildProgram(program, num_devices, devices, NULL, NULL,
NULL);
  if (err != CL_SUCCESS)
  {
      // Determine the reason for the error
      char buildLog[16384];
      clGetProgramBuildInfo(program, devices[0],
CL_PROGRAM_BUILD_LOG,
                            sizeof(buildLog), buildLog, NULL);

      std::cerr << "Error in program: " << std::endl;
      std::cerr << buildLog << std::endl;

      return err;
```

```
  }

  // Write the binaries
  write_binaries(program, num_devices, platform_idx,
outputBinaryName);

  return err;
}

void compile_all(const char *src, size_t src_size, const char *
outputBinaryName ) {
  cl_uint i;

  // Get the platform list
  cl_platform_id *platforms = NULL;
  cl_uint num_platforms = 0;
  if (get_platform_list(&platforms, &num_platforms) != CL_SUCCESS) {
    return;
  }

  // For each platform compile binaries for each devices
  for (i = 0; i < num_platforms; ++i) {
    // Compile for each devices
    cl_uint num_devices = 0;
    cl_int err = compile_program(&num_devices, src, src_size,
platforms[i], i, outputBinaryName );

    // Print the result
    char *platform_name = get_platform_info(platforms[i],
CL_PLATFORM_NAME);
    printf("PLATFORM [%s]  -->  %s (%u)\n",
           (platform_name ? platform_name : ""),
           ((err == CL_SUCCESS) ? "SUCCESS" : "FAILURE"),
           (unsigned)num_devices);
    if( err ){
      cerr << "[Error code]" << err << endl;
      exit(-1);
    }
    fflush(stdout);
    free(platform_name);
  }

  // Free the platform list
  free_platform_list(platforms, num_platforms);
}

int main(int argc, char **argv) {
  // Check the command line option
  if (argc < 3) {
    cerr << "USAGE: opencl-compiler [SOURCE] [OUTPUT NAME]\n";
    exit(EXIT_FAILURE);
  }

  const char * filename = argv[1];
```

```
  const char * output_fn = argv[2];

  // Read the source file
  char *src = NULL;
  size_t src_size = 0;
  if (read_file(&src, &src_size, filename) != 0) {
    cerr << "ERROR: Failed to read:" << filename << endl; return -1;
  }

  // Compile binaries for each platforms and devices
  compile_all(src, src_size, output_fn);

  // Free the source file
  free(src);

  // Get the platform list
  cl_int err = CL_SUCCESS;
  cl_platform_id *platforms = NULL;
  cl_uint num_platforms = 0;
  if (get_platform_list(&platforms, &num_platforms) != CL_SUCCESS) {
    cerr << "ERROR: Failed to get_platform_list" << endl; return -1;
  }

  // Get the device list from the first platform
  cl_device_id* devices = NULL;
  cl_uint num_devices = 0;
  get_device_list(&devices, &num_devices, platforms[0]);

  cl_context ctx = clCreateContext(NULL, num_devices, devices, NULL,
                                   NULL, &err);
  if (err != CL_SUCCESS) {
    cerr << "fail to create contenxt" << endl; return -1;
  }

  // Create a new program
  cl_program cl_progLoadedFromBinary;

  // Load the kernel binary
  cl_progLoadedFromBinary = CreateProgramFromBinary(ctx, *devices,
output_fn);
  if ( !cl_progLoadedFromBinary ){
    cerr << "Fail to create program" << endl; return -1;
  }else{
    cout << "Program created from binary file " << output_fn <<
endl;
  }

  return 0;
}
```

```
#include <sys/time.h>
#include <time.h>
#include <random>

#include "tensorflow/core/public/session.h"
#include "tensorflow/core/graph/default_device.h"

using namespace tensorflow;
using namespace std;

int main(int argc, char* argv[]) {

    if( argc != 8 ){
      cerr << "expected 2 arguments [rowA] [colA] [rowB] [colB]
[TransA] [TransB] [Num of Runs]" << endl;
      exit(1);
    }

    // Random generator
    std::random_device rd;
    std::default_random_engine gen =
std::default_random_engine(rd());
    std::normal_distribution<> dis{0,5};

    // Timers
    struct timeval start, end;

    string graph_definition = "matmul.pb";
    Session* session;
    GraphDef graph_def;
    SessionOptions opts;
    vector<Tensor> outputs; // Store outputs
    TF_CHECK_OK(ReadBinaryProto(Env::Default(), graph_definition,
&graph_def));

    // Set graph options
    graph::SetDefaultDevice("/cpu:0", &graph_def);

    // create a new session
    TF_CHECK_OK(NewSession(opts, &session));

    // Load graph into session
    TF_CHECK_OK(session->Create(graph_def));

    // Matrix transpose option before matrix multiplication
    int transA = atoi( argv[5] );
    int transB = atoi( argv[6] );

    // Matrix size
    int rowA = atoi( argv[1] );
    int colA = atoi( argv[2] );
    int rowB = atoi( argv[3] );
    int colB = atoi( argv[4] );
    int rowC = (transA == 1) ? colA : rowA;
```

```
    int colC = (transB == 1) ? rowB : colB;

    // Number of runs
    int num_runs = atoi( argv[7] );

    // Tensorflow Tensor initializaiotn
    Tensor TensorA (DT_FLOAT, TensorShape({ rowA, colA }));
    Tensor TensorB (DT_FLOAT, TensorShape({ rowB, colB }));
    float * TensorC = (float*)malloc( rowC * colC * sizeof(float) );

    // Matrix initializaiotn
    auto TensorAMatrix = TensorA.tensor<float, 2>();
    for( int i = 0 ; i < rowA ; i ++ ){
      for( auto j = 0 ; j < colA ; j ++ ){
        TensorAMatrix(i, j) = dis(gen);
      }
    }
    auto TensorBMatrix = TensorB.tensor<float, 2>();
    for( int i = 0 ; i < rowB ; i ++ ){
      for( auto j = 0 ; j < colB ; j ++ ){
        TensorBMatrix(i, j) = dis(gen);
      }
    }

    LOG(INFO) << ">>> [TF] Starting " << num_runs << " TF MatMul
runs...";
    // Start timer
    gettimeofday(&start, NULL);

    for (int r=0; r<num_runs; r++) {
      // Compute matrix multiplaction result using TF
      TF_CHECK_OK(session->Run({{"x", TensorA}, {"y", TensorB}},
{"matmul"},
        {}, &outputs)); // Get cost
    }
    auto tf_res = outputs[0].matrix<float>();
    // cout << "TF result: \n" << tf_res << endl;

    // Stop timer
    gettimeofday(&end, NULL);

    double interval = ( end.tv_sec * 1.0e6 + end.tv_usec ) -
      ( start.tv_sec * 1.0e6 + start.tv_usec );
    double runtime = interval / num_runs;
    std::cerr << ">>> Done: took " << runtime << " us per run";
    std::cout << runtime << endl;

    // Compute matrix multiplaction result using Eigen
    auto TensorAEigenMap = Eigen::Map<Eigen::Matrix<
      float,           /* scalar element type */
      Eigen::Dynamic,  /* num_rows is a run-time value */
      Eigen::Dynamic,  /* num_cols is a run-time value */
      Eigen::RowMajor  /* tensorflow::Tensor is always row-major */
>>(
```

```
        TensorA.flat<float>().data(),  /* ptr to data */
        rowA,           /* num_rows */
        colA            /* num_cols */);

    auto TensorBEigenMap = Eigen::Map<Eigen::Matrix<
      float,           /* scalar element type */
      Eigen::Dynamic,  /* num_rows is a run-time value */
      Eigen::Dynamic,  /* num_cols is a run-time value */
      Eigen::RowMajor  /* tensorflow::Tensor is always row-major */
>>(
        TensorB.flat<float>().data(),  /* ptr to data */
        rowB,           /* num_rows */
        colB            /* num_cols */);

    auto eigen_res = Eigen::Map<Eigen::Matrix<
      float,           /* scalar element type */
      Eigen::Dynamic,  /* num_rows is a run-time value */
      Eigen::Dynamic,  /* num_cols is a run-time value */
      Eigen::RowMajor  /* tensorflow::Tensor is always row-major */
>>(
        TensorC,  /* ptr to data */
        rowC,           /* num_rows */
        colC            /* num_cols */);

    cout << ">>> [Eigen] Starting " << num_runs << " Eigen MatMul
runs...";
    // Start timer
    gettimeofday(&start, NULL);

    if( transA == 1 && transB == 1 ){
      eigen_res = ( TensorAEigenMap.transpose() *
TensorBEigenMap.transpose() );
    }else if( transA == 1 && transB == 0 ){
      eigen_res = ( TensorAEigenMap.transpose() * TensorBEigenMap );
    }else if( transA == 0 && transB == 1 ){
      eigen_res = ( TensorAEigenMap * TensorBEigenMap.transpose() );
    }else if( transA == 0 && transB == 0 ){
      eigen_res = ( TensorAEigenMap * TensorBEigenMap );
    }

    // cout << "Eigen result: \n" << eigen_res << endl ;

    // Stop timer
    gettimeofday(&end, NULL);
    interval = ( end.tv_sec * 1.0e6 + end.tv_usec ) -
      ( start.tv_sec * 1.0e6 + start.tv_usec );
    runtime = interval;
    std::cout << ">>> Done: took " << runtime << " us per run";
    std::cout << runtime << endl;

    cout << "Checking results ...\n";

    double accu_err = 0;
    double signErrCount = 0;
```

```
    double valueErrCount = 0;
    for( auto row = 0 ; row < rowC ; row ++ )
    {
      for( auto col = 0 ; col < colC ; col ++ ){
        float tmp = abs( tf_res(row, col) - eigen_res(row, col) );
        accu_err += tmp;
        if(  tf_res(row, col) * eigen_res(row, col) < 0 ){
          // cout << "(" << row << "," << col << ") sign err, tf_res
" << tf_res(row, col) << " eigen_res " << eigen_res(row, col) <<
endl;
          signErrCount++;
        }
        else if( tmp > 1 ){
          // cout << "(" << row << "," << col << ") val err, tf_res
" << tf_res(row, col) << " eigen_res " << eigen_res(row, col) <<
endl;
          valueErrCount++;
        }
      }
    }
    cout << "err per unit: " << accu_err/(rowC*colC) << ",
signErr(%) " << signErrCount/(rowC*colC) << ", valueErr(%) " <<
valueErrCount/(rowC*colC) << endl;

    free(TensorC);

    return 0;
}
```

G. OpenCL memory bandwidth test source code

```
#include "CL/cl.h"
#include "Timer.h"
#include "clMemTester.h"
#include <iostream>

// clMemTester constructor
clMemTester::clMemTester(int num){
  numTests = num;
}

// Init OpenCL objects
cl_int clMemTester::init()
{
  // OpenCL error code init
  err = CL_SUCCESS;

  // Query platforms
  err = clGetPlatformIDs(1, &platform, NULL);
  if( err != CL_SUCCESS )
    return err;

  // Query devices
  err = clGetDeviceIDs(platform, CL_DEVICE_TYPE_ALL, 1, &clDevice,
NULL);
  if( err != CL_SUCCESS )
    return err;

  // Create context
  clCtx = clCreateContext(NULL, 1, &clDevice, NULL, NULL, NULL);

  // Create command clQueue
  clQueue = clCreateCommandQueue(clCtx, clDevice, 0, NULL);

  // Timer init
  Timer timer = Timer();

  return CL_SUCCESS;
}

// Release all OpenCL related resourcse
cl_int clMemTester::clEnd(){
  clReleaseCommandQueue(clQueue);
  clReleaseContext(clCtx);
  return CL_SUCCESS;
}

// Host to device memory bandwidth test
cl_int clMemTester::HostToDevice( unsigned long int numBytes )
{
  // Create host buffer
  char * hostBufPtr = new char [ numBytes ];
  for ( auto i = 0; i < numBytes; i++ )
  {
      hostBufPtr[i] = (i & 0xff);
```

```
  }

  // err code init
  err = CL_SUCCESS;

  // Create device buffer
  cl_mem deviceBuffer = clCreateBuffer( clCtx, CL_MEM_READ_WRITE,
numBytes, NULL, &err );
  if ( err != CL_SUCCESS )
  {
      std::cerr << "clCreateBuffer fail with code " << err;
      delete [] hostBufPtr;
      return err;
  }

  clFinish( clQueue );

  timer.start();

  // Write host -> device
  for ( size_t i = 0; i < numTests; i++ )
  {
      // Asynchronous write
      err = clEnqueueWriteBuffer( clQueue, deviceBuffer, CL_FALSE,
0, numBytes,
        hostBufPtr, 0, NULL, NULL );
      if (err != CL_SUCCESS )
      {
          std::cerr << "Error writing device buffer";
          clReleaseMemObject( deviceBuffer );
          delete [] hostBufPtr;
          return err;
      }
  }

  // Finish any outstanding writes
  clFinish( clQueue );

  computeBandwidth( numBytes, timer.read_us() );
  delete [] hostBufPtr;
  clReleaseMemObject( deviceBuffer );
  return CL_SUCCESS;
}

// Device to host memory bandwidth test
cl_int clMemTester::DeviceToHost( unsigned long int numBytes )
{
  // Create host buffer
  char * hostBufPtr = new char [ numBytes ];
  for ( auto i = 0; i < numBytes; i++ )
  {
      hostBufPtr[i] = (i & 0xff);
  }
```

```
  // err code init
  err = CL_SUCCESS;

  // Copy the contents of the host buffer into a device buffer
  cl_mem deviceBuffer = clCreateBuffer( clCtx,
    CL_MEM_READ_WRITE | CL_MEM_COPY_HOST_PTR, numBytes, hostBufPtr,
&err );
  if ( err != CL_SUCCESS )
  {
      std::cerr << "clCreateBuffer fail with code " << err;
      delete [] hostBufPtr;
      return err;
  }

  clFinish( clQueue );

  timer.start();

  // Read from device -> host
  for ( size_t i = 0; i < numTests; i++ )
  {
      // Asynchronous read
      err = clEnqueueReadBuffer( clQueue, deviceBuffer, CL_FALSE, 0,
numBytes,
        hostBufPtr, 0, NULL, NULL );
      if (err != CL_SUCCESS )
      {
          std::cerr << "Error writing device buffer";
          clReleaseMemObject( deviceBuffer );
          delete [] hostBufPtr;
          return err;
      }
  }

  // Finish any outstanding writes
  clFinish( clQueue );

  computeBandwidth( numBytes, timer.read_us() );
  delete [] hostBufPtr;
  clReleaseMemObject( deviceBuffer );
  return CL_SUCCESS;
}

// Device to device memory bandwidth test
cl_int clMemTester::DeviceToDevice( unsigned long int numBytes )
{
  // Create host buffer
  char * hostBufPtr = new char [ numBytes ];
  for ( auto i = 0; i < numBytes; i++ )
  {
      hostBufPtr[i] = (i & 0xff);
  }

  // err code init
```

```
  err = CL_SUCCESS;

  // Copy the contents of the host buffer into a device buffer
  cl_mem deviceBufferSrc = clCreateBuffer( clCtx,
    CL_MEM_READ_WRITE | CL_MEM_COPY_HOST_PTR, numBytes, hostBufPtr,
&err );
  if ( err != CL_SUCCESS )
  {
      std::cerr << "clCreateBuffer fail with code " << err;
      clReleaseMemObject( deviceBufferSrc );
      delete [] hostBufPtr;
      return err;
  }

  // Create another device buffer to copy into
  cl_mem deviceBufferDst = clCreateBuffer( clCtx, CL_MEM_READ_WRITE,
numBytes,
    NULL, &err );
  if ( err != CL_SUCCESS )
  {
      std::cerr << "clCreateBuffer fail with code " << err;
      clReleaseMemObject( deviceBufferDst );
      delete [] hostBufPtr;
      return err;
  }

  clFinish( clQueue );

  timer.start();

  // Copy from device -> device
  for ( size_t i = 0; i < numTests; i++ )
  {
      // Asynchronous write
      err = clEnqueueCopyBuffer( clQueue, deviceBufferSrc,
deviceBufferDst,
        0, 0, numBytes, 0, NULL, NULL );
      if (err != CL_SUCCESS )
      {
          std::cerr << "Error copying device buffer";
          clReleaseMemObject( deviceBufferSrc );
          clReleaseMemObject( deviceBufferDst );
          delete [] hostBufPtr;
          return err;
      }
  }

  // Finish any outstanding writes
  clFinish( clQueue );

  computeBandwidth( numBytes, timer.read_us() );
  delete [] hostBufPtr;
  clReleaseMemObject( deviceBufferSrc );
  clReleaseMemObject( deviceBufferDst );
```

```
  return CL_SUCCESS;
}

// Memory bandwidth calculator
void clMemTester::computeBandwidth(size_t numOfBytes, const double&
time_us){

  double MB = numOfBytes / (1024*1024);
  printf("%.2f\n", MB * numTests * 1e6 / time_us / 1024);
}
```