Deep Learning on Mobile Platforms with GPUs

JI Zhuoran 3035139915

The University of Hong Kong

The Final Report Submitted for Review

15 April 2018

Acknowledgements

I sincerely thank my supervisor, Prof. Wang for helping me with all aspects of the project. I would also like to express gratitude to those Open-Source Datasets Owner, for their effort and work on these valuable datasets. I am grateful for help received from Dr. Tam for his valuable comments. Last but not least, I want to express my appreciation towards all members in High-Performance Computing Research Group of The University Hong Kong, for their kind supports. I am sincerely thankful for all the effort, favor, feedback and suggestions from them.

Abstract

Breakthroughs in the fields of deep learning and mobile processor chips are radically changing the way we use our smartphones. However, there are few studies of optimizing mobile deep learning frameworks for inference speed and memory reduction, which prohibits further usage of deep learning on mobile platforms. In this work, we presented the design and implementation of MobileDL, a toolkit that is exclusively dedicated to mobile devices. MobileDL significantly accelerated the inference stage of convolution neural network with the help of three novel methodologies: (1) Convolutional Neural Network Compression; (2) Zero-Copy and (3), Half Precision Computation Supporting. Among all these optimization method, Convolutional Neural Network Compression is the core part of the whole project, which includes two compression methods named Channels-Oriented K-Means Clustering Compression and Mask-Based Neural Network Recovery. Experiments on several famous benchmarks demonstrated about $3\times$ *speed-up with merely* 1% *loss of classification accuracy. Additionally, MobileDL also achieves significant memory usage reduction, which is critical to mobile and embedding devices. With MobileDL, more innovative and fascinating mobile applications will be turned into reality.*

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Background

In recent years, a great success of General Purpose Graphics Processor Units (GPGPUs) in massive computing tasks has been witnessed. This achievement encourages processor manufactures to improve general computing capabilities of GPUs. Nowadays, programmable GPUs are also available on mobile devices, such as smartphones, autopilot cars, and IoT devices, which leads to significant performance boost and substantial energy reduction for massively mobile computation tasks [7].

Deep learning has also drawn significant attention recent years, especially in computer vision [8], speech recognition [9], and natural language processing tasks [10]. Almost all the of recent successful systems in these areas are built based on deep neural networks. However, even these technologies are critical to many mobile-phone apps, only few of them take advantage of deep learning techniques [11]. This situation is caused by limited computation ability and memory space of mobile devices. Additionally, as these networks grow more and more complicated, computation is also increasing exponentially, that

makes deployment even more intractable [12].

1.2 Current Status

The mainstream of these successful attempts of mobile deep learning usage is based on cloud computing [11], which has several drawbacks, such as affecting privacy confidentiality, no real-time guarantee, and network overhead. However, porting deep learning framework to mobile or embedded devices is not trivial and is relatively under-studied especially on GPUs. There are a few of successful attempts in using mobile CPU for local execution. It seems that CPUs present an attractive potential solution because they are available on almost all mobile devices. However, CPUs will drain batteries in few hours if not few minutes, while most apps keep doing inference during executing or even on background. As a result, CPU solution is not suitable for battery powered devices.

1.3 Objective

This project will introduce the MobileDL, a deep learning toolkit that executed locally on mobile GPUs with reasonable speed and battery consumption. Instead of porting current frameworks directly, this toolkit is highly customized for mobile GPUs by taking computation, memory limitation and power consumption into consideration. Though MobileDL is customized for mobile devices, it is still a cross-platform software, in other words, MobileDL is executable on any platforms as long as Open Computing Language (OpenCL) is supported, as no assumption of specific GPU architectures is make. However, beyond code-level modifications, it offers three novel opti-

mization methods, namely: (1) Zero-Copy; (2) Convolution Neural Network Compression and (3) Half Precision Support. Among all these optimization method, Convolutional Neural Network Compression is the core part of the whole project. Through these three optimization, MobileDL offers a fast solution for deep learning on mobile platforms with GPUs.

1.4 Outline

The following parts of this report have been organized as follows: first introducing the theoretical background (§ II); next discussing several related works (§ III); after that, explaining the software architecture of Caffe, from which our framework is built (§ IV); then presenting the core part of this report, neural network compression and the experiment results (§ V, VI); next, optimization method named half precision matrix multiplication and unified memory between CPUs and GPUs are discussed (§ VII, VIII), and finally, application example and a brief summary is presented (§ IV).

Chapter 2

Theoretical Backgrounds

This chapter will first explain basic concepts related to mobile GPUs, convolutional neural network and K-means clustering. Then, our platforms, Adreno 540 will be discussed in detail.

2.1 Preliminaries

2.1.1 Mobile GPUs

Mobile GPUs have become increasingly powerful, which pushes forward the general computing technology for mobile devices over the past few years [13]. However, only a few papers discussed the general computing capabilities of mobile GPUs. Experience on desktop GPUs is not applicable on mobile GPUs, as design criteria of mobile GPUs is different from desktop GPUs. First of all, as mobile GPUs are usually powered by batteries, they are generally with lower frequency and much fewer cores [14]. Additionally, as most mobile GPUs are integrated into SoCs (System on Chip), graphics memories are not available [15][16] and accessing external memory will lead to much lower memory bandwidth. Last but not least, there are plenty of mobile GPU families, such as Qualcomm's Adreno family [15], Mali family [16], and

NVIDIA Tegra family [17], leading to varies of mobile GPU architectures.

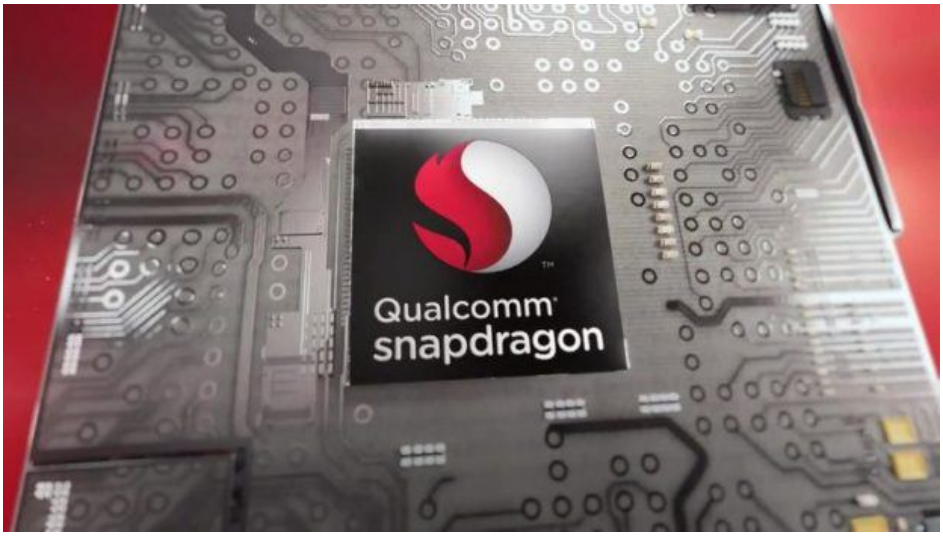


Figure 2.1: Snapdragon 835

2.1.2 Convolutional Neural Networks

Convolutional Neural Networks (CNNs), which is composed of three major layers: convolution layers, pooling layers, and fully connected layers are the state-of-the-art neural networks for vision and image related tasks [18].

The core operations in convolution layers are 2-dimensional sliding-window convolutions with a 2-dimensional convolution filter. Each convolution filter is related to one input channel and one output channel. Each convolution filter first convolves with its corresponding input activation plane and then be accumulated to its output activation plane [19]. For a convolution layer with C input channels, K output channels, a $R \times S$ element filter is applied over a $W \times H$ element input channel to produce a $W \times H$ output activation plane. The overall time complexity is $\mathcal{O}(C \times K \times R \times S \times W \times H)$.

For most CNNs, convolution layers dominate execution time in the inference stage. For example, for a typical neural network described in [20], on our platform: Snapdragon 820 development board, all its five convolution layers take 81.7% of the forwarding time, while all other layers only take the reminding 18.3%. The reason why convolution layers consume so much time is that, the order of time complexity of convolution layers is much higher than other layers. As discussed in the previous paragraph, the time complexity of convolution layers is $\mathcal{O}(C \times K \times R \times S \times W \times H)$, in other words, in each convolution layer, there are $C \times K$ convolution filters, and for each filter, $\mathcal{O}(R \times S \times W \times H)$ computation is needed, where $C \times K$ is usually more than 32×32 in state-of-the-art CNNs [20][21][22].

The time complexity of convolution layers makes local execution intractable, however, Denil et al. demonstrated that there are huge redundancies in neural networks [23]. They achieved an accurate prediction of all parameter within a layer by only a small subset (about 5%), which implies neural networks can be heavily compressed. However, these redundancies are necessary during the train stage as deeper neural networks provide larger capacities for functional approximation. Their work inspires us to apply K-means clustering method to explore the redundancies of the parameter space.

2.1.3 K-means clustering

K-means clustering is a method for vector quantization [24], originally from data mining, that is also popular for data clustering. The basic idea of K-means clustering is partitioning n vectors into k clusters in which each

vector is represented by the center of the whole cluster.

K-means clustering is an NP-hard problem [25]. However, many efficient heuristic algorithms converge quickly [26][27], especially with initializations that are close to the final result. As K-means clustering is applied in every iteration during training in MobileDL and parameters are updated smoothly in gradient descent method [28], the clustering result in the last iteration should close to clustering mean in this iteration. These two properties make it possible to use the previous result as initialization, which makes K-means clustering extremely efficient, so the overhead introduced by our approach is not significant.

2.2 Adreno 540

Snapdragon [29] is one of the most powerful and widely used mobile processors in today's mobile phones and the Internet of Things systems. Snapdragon is a System on Chip (SoC) processor that integrates CPU, GPU, DSP and other specialized processing units. Mainly used for UI rendering, Adreno GPUs in Snapdragon SoC, especially for Adreno 5XX, are also one of the state-of-the-art mobile general purpose processors that can better handle massive computation tasks.

2.2.1 Specification

There is no official hardware specification for Adreno GPU. All information are got by OpenCL program, which is shown in table 2.1. Compare to the difference of computation power between Desktop's GPUs and Mobile

GPU Model	Adreno 540
Global Memory Size	3001430016 Bytes = 2.78 GB
Global Cache Size	131072 Bytes = 128 KB
Cache Line Size	64 Bytes
Local Memory Size	32768 Bytes = 32 KB
Constant Buffer	65536 Bytes = 64 KB
Number of Compute Units	4
Number of ALUs	256
Max Work Groups	1024
Max Work Items	1024
Unified Memory Supporting	Native
Half Precision Supporting	Native

Table 2.1: Hardware Specification of Aderno 540

GPUs, the gap of memory latency and bandwidth is extremely significant, as host memory is usually limited and extremely slow, for example, in Table 2.2, memory capability and bandwidth of several state-of-the-art mobile and desktop devices are shown. It can be seen that the capability of desktop GPUs memory is usually more than 3× larger than host memory of mobile devices, much worse, as host memory is shared by other computation units, the actual memory available to the GPU is even less. Furthermore, the bandwidth gap is even more significant, as desktop GPU's bandwidth is about 25× wider than that of mobile devices.

Manufacture	Model	Platform	Capacity	Bandwidth
XIAOMI	Mi6 [30]	Mobile	6 GB	25.6 GB/s
APPLE	iPhone X [31]	Mobile	3 GB	28.3 GB/s [32]
HUAWEI	MATE 10 [33]	Mobile	6 GB	21.3GB/s
NVIDIA	1080 Ti [34]	Desktop	11 GB	480 GB/s
AMD	Vega 64 [35]	Desktop	8 GB	483.8 GB/s

Table 2.2: Comparison on various of GPUs

2.2.2 Memory Architecture [1]

Figure 2.2 shows the memory architecture of Adreno 5XX GPUs family. Different with dedicated GPUs on desktops, Adreno GPUs share host memory with other computation units, such as CPUs and DSP, which is called global memory in OpenCL model (the green part). Every data accessed by computing units must be stored in the L2 cache on Adreno GPUs, which is 128 KB in total and is referred to global memory data cache (the red one). Global memory data cache is 10× faster than global memory. However, they are both off-chip memories. For local memory, the blue parts in figure 2.2, is hundreds of times faster than the previous two. For Adreno 540, there is 32 KB local memories, which is equally divided into four parts, as there are four compute units, so for each computation unit, at most 8 KB local memory can be used. This indicates us that the working window size of highly reusable data should be restricted within 8 KB and declared as local variables to get better performance.

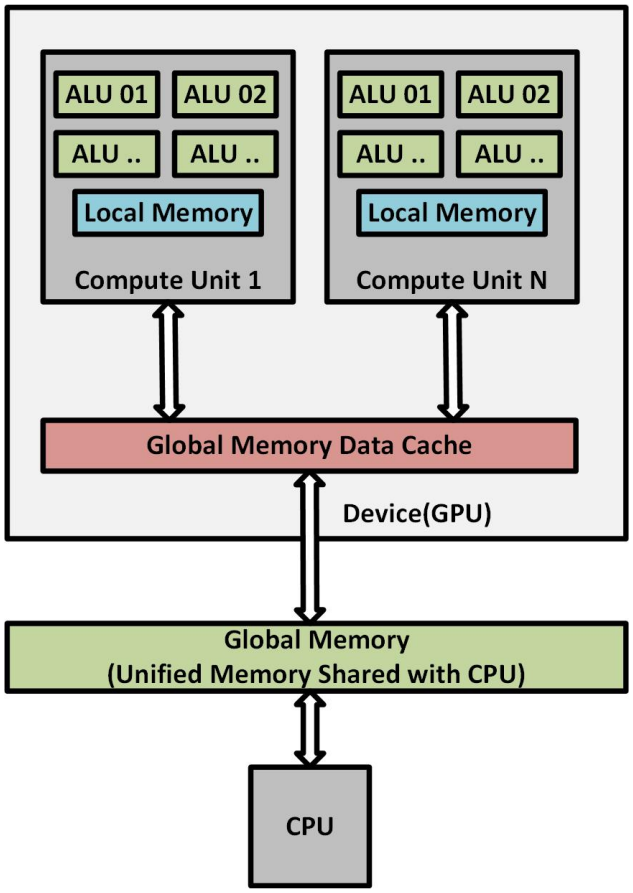


Figure 2.2: Memory architecture of Adreno 5XX family

Chapter 3

Literature Reviews

3.1 Deep Learning on Mobile Devices

Almost all popular deep learning frameworks, such as Caffe [8], Tensorflow [36], Torch7 [37], Caffe2Go [38], Deeplearning4j [39], support Android platforms, and Shiro [40] took an important first step towards porting deep learning frameworks to mobile devices and achieved Cifar-10 recognition on Android devices in a reasonable time. However, only a few of these frameworks adjusted source codes for performance optimization for mobile devices. Even worse, all of them only provide CPU-based solutions for Android platforms, which is not feasible as discussed in § I.

DeepEye [41] demonstrated a device that is capable of executing several state-of-the-art deep vision models with nearly 17 hours battery life. DeepX [11] then proposed a decomposition method which split monolithic networks into unit-blocks of various types, significantly reduces the latency of full connected layers. Both of these two works demonstrated notable speedup and power-saving by taking mobile hardware-characteristics into consideration when designing the framework.

CNNdroid [42] proposed an Android GPU-accelerated library, which specifically designed and optimized for inference-only tasks on Android-based mobile devices. Then DeepMon [43] showed early evidence that mobile device can handle large DNNs, and devised a suite of optimization techniques to reduce the processing latency.

Different from these previous works, MobileDL is a deep learning framework highly customized for less-powerful mobile devices. Also, MobileDL supports OpenCL, which enables MobileDL running on heterogeneous SoCs, especially on power-efficiency computation units, such as GPUs. Finally, different from CNNdroid and DeepMon, MobileDL is more aggressive, as little accuracy loss is permitted.

3.2 Neural Network Compression

After Denial et al. [23] proved the redundancies of neural networks, several CNN compression approaches have been proposed. Denton et al. [44] showed an early successful attempt of compressing the fully-connect layer by applying truncated singular value decomposition with insufficient loss of the prediction accuracy. Then Gong et al. [12] exploited different vector quantization methods for neural network compression. Different with these previous works, which focus on reducing storage of network parameters, our approach focuses on computation reduction.

Jaderberg et al. [45] presented the speedup penitential of convolutional

neural networks by low-rank decomposition of convolution tensors, they achieved about 2× speedup on desktop CPUs. Then Lebedev et al. [46] demonstrated that 8.5× is obtained by CP-decomposition on a full convolution tensor with only minor accuracy drop (from 91% to 90%). Kim et al. [47] applied these compression techniques discussed above for fast and low-power mobile deep learning applications. They tested AlexNet [20], VGG-16 [21], and GoogleLeNet [22] in aspects of both energy consumption and execution time, and proved that these complexity state-of-the-art neural networks executed efficiently on mobile devices after compression. MobileDL extends these works by taking memory divergence and parallelism into consideration to make compressed neural networks efficiently executed on GPUs.

3.3 Quantized Convolution Neural Network

Wu et al. [48] proposed a unified framework for CNNs, named Quantized CNN(Q-CNN). Q-CNN quantize convolution tensors along the dimension of the output channels. By splitting the weighting matrix into several sub-matrices and learning a code-book on each of them, each sub-matrix is quantized into a smaller matrix with a code-book. Compressed outputs are computed by convolution between smaller matrices and original inputs. Then desired outputs are reconstructed by searching the compressed outputs and the code-books. MobileDL differs from Wu et al.'s work in that MobileDL takes advantage of massively computation units such as GPUs, so memory divergence introduced by Q-CNN should be removed. On the other hand, MobileDL does not introduce any memory storage overhead as neither code-book nor sub-matrix needs to be stored. Finally, MobileDL modifies

fine-tuning method to drag the related convolution filters close to each other, by which accumulated errors are further reduced.

Chapter 4

Caffe-Moble-OpenCL: A Detailed Explanation

There are several famous deep learning frameworks, such as Caffe [8], Tensorflow [36], Deeplearning4j [39], and MXNet [49]. Among all these frameworks, Caffe is chosen as the base for this project for Three main reasons. First, it is well-know that, Caffe is the only deep learning framework that officially supports OpenCL, which is the only general computation API available on non-NVIDIA mobile devices. Secondly, Caffe is powerful enough to handle almost all kinds of neural networks, but not too complicated to be modified. Last but not least, the software architecture design (see Figure 4.1) is pretty good and clear, which makes the modification and implementation easier.

4.1 Backends

There are mainly four different Backends in the Caffe-OpenCL, which are Caffe-Greentea-GPU, Caffe-CPU, Caffe-Greentea-BLAS and Caffe-LIBDNN. Instead of discussing the implementation of all layers, in this section, the convolution layer, which is the most typical layer, is explained in detail.

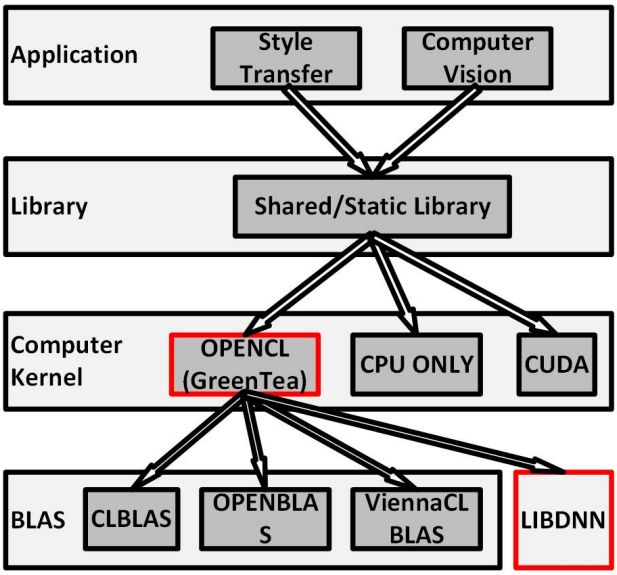


Figure 4.1: Software Architecture of Caffe

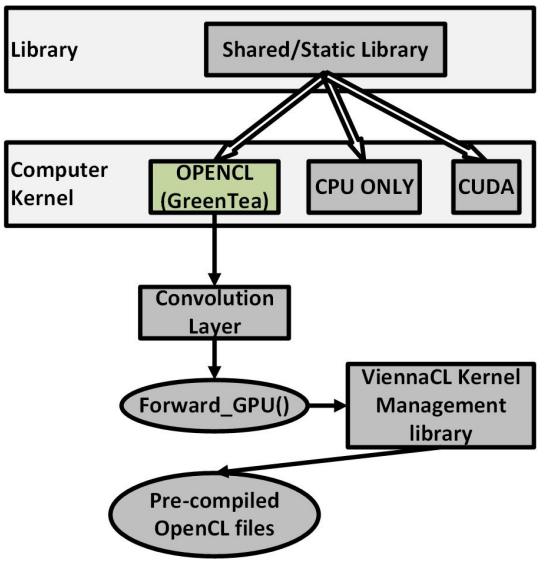


Figure 4.2: Caffe-Greentra-GPU

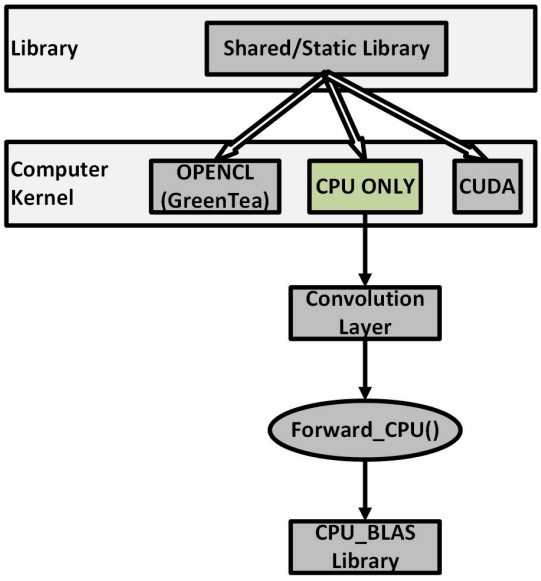


Figure 4.3: Caffe-CPU

4.1.1 Caffe-Greentea-GPU

For Caffe-Greentea-GPU (figure 4.2), it is the most trivial implementation. The execution logic is quite straightforward, in convolution layers, `forward_gpu()` will call ViennaCL [50] kernel management library to get and execute the pre-compiled OpenCL functions. This implementation is no

longer used due to its poor performance.

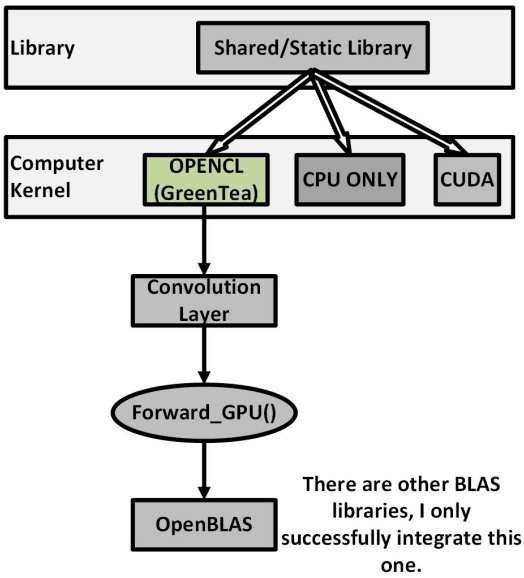


Figure 4.4: Caffe-Greentea-BLAS

4.1.2 Caffe-CPU

Caffe-CPU (figure 4.3) is a so-called `CPU_ONLY` implementation. As only the CPU is used for computation, the execution logic is even more straightforward. As far as I know, almost all popular deep learning frameworks choose similar implementations for their mobile version [8][36][39], as it is more portable. However, the performance is relatively poor in Snapdragon 835, which is unexpected, as CPU of Snapdragon is as powerful as GPU, even for massively computation tasks.

4.1.3 Caffe-Greentea-BLAS

For the third one, Cafee-Greentea-BLAS (figure 4.4), there are several sub-implementations depends on different BLAS functions. When convolution layers call `forward_gpu()` function, BLAS function is called as same as Caffe-

CPU, except it is the GPU that executes computations. However, this final year project will not focus on BLAS library improvement.

4.1.4 Caffe-LIBDNN

In this final year project, all optimization are based on the fourth implementation, which is Caffe-LIBDNN (figure 4.5), as it is the state-of-the-art solution no matter in execution time or memory usage. However, the execution logic is quite complicated compared to other implementations. Instead of `conv_layer`, `libdnn_conv_layer` is constructed, in which `reshape()` will call libdnn kernel generator to generate and compile the OpenCL kernels at runtime. The advantage of run-time compilation is that most of the runtime-determined parameters that needed to be passed as arguments in other implementations can be defined as constant variables, by which the compiler can better optimize the OpenCL program. After that, the execution logic is as same as Caffe-Greentea-GPU.

4.1.4.1 LIBDNN

This section focuses on the reasons and benefits of integrating LIBDNN into MobileDL. LIBDNN is a universal convolution library, which is implemented by Shiro [40]. Though LIBDNN is not our original work, it is analyzed as it improves MobileDL's performance significantly. The following part of this sub-section is organized as follow: first, analyzing the memory access overhead in Caffe, next, exploring why Caffe's performance drops significantly on mobile platforms compare to desktops. Finally, explaining how LIBDNN solved this problem.

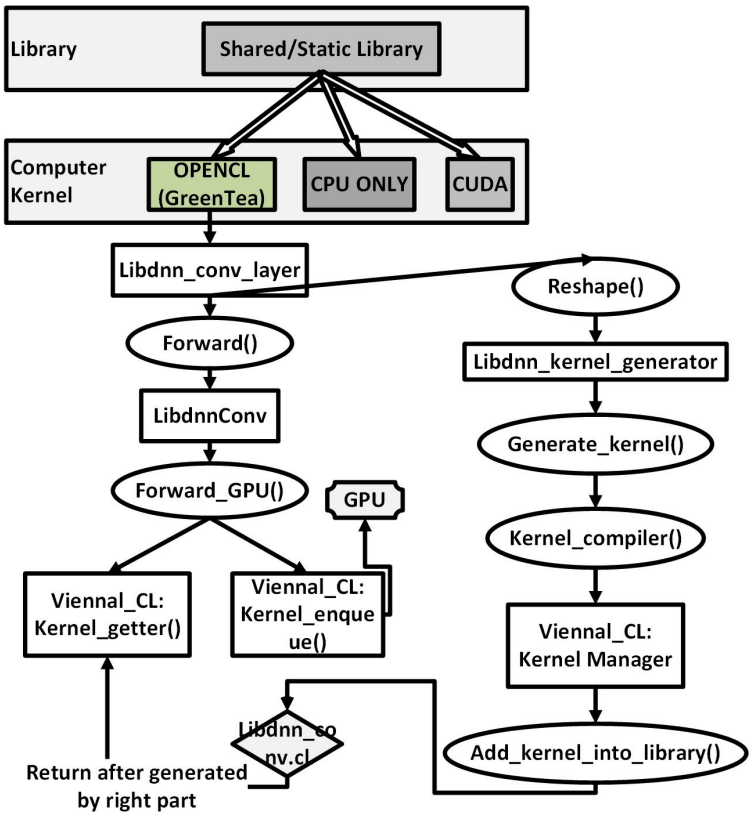


Figure 4.5: Caffe-LIBDNN

4.1.4.2 Memory overhead in Caffe

The dominant memory overhead of convolution layer in original Caffe is that it converted convolution computation to matrix multiplication by expanding images, to take advantage of the existing optimization method of matrix multiplication. For a $W \times H$ image and $R \times S$ convolution filter, for any pixel in this image, the $R \times S$ filter was convolved with the corresponding $R \times S$ sub-image of this pixel. In Caffe, the whole sub-image was reshaped to a $(R * S) \times 1$ column vector, then saved in the new expaned matrix. As a result, the size of the expanded matrix became $R \times S \times W \times H$. Since the new matrix was stored in the memory during computation, $R \times S$ times memory storage and access were needed.

4.1.4.3 Reason for performance drop

The memory overhead is not a problem on desktop GPUs, but for mobile GPUs, which have only limited slow shared memory, the overhead becomes intolerable. As discussed in the chapter § II, no mobile GPU has individual graphics memory integrated. Instead, mobile GPUs share host memory with CPUs and other computation units.

To overcome this problem, LIBDNN has been integrated into MobileDL. With the help of LIBDNN, the construction of expanded matrix now is performed lazily. More specifically, the original image is loaded into GPUs on the fly, and the expansion of sub-images is put off until it is needed. Furthermore, the expanded sub-image will be discarded immediately after access and be re-computed if it is needed again.

4.1.4.4 Performance Discussion

Though there are $\frac{W \times H}{WPTW \times WPTH} \times$ computation overheads compared to the original method, where $WPTW$ and $WPTH$ are work-per-thread along W and H axis respectively. It is a reasonable tradeoff, as memory access is much more expensive than computation, and $\frac{W \times H}{WPTW \times WPTH}$ is less than 8 for almost all typical convolution neural networks.

Empirical results indicated that up to 15× speedup was achieved for most of the state-of-the-art convolution neural networks on our Snapdragon development board. As LIBDNN is not our original work, and the method is quite

trivial and well-studied, the evaluation of LIBDNN will not be included in this report. Further information is available on the author's GitHub [40].

4.1.5 Performance of different Backends

All experiments are performed on Snapdragon 820 development board and Android 6. The benchmark we choose is the most typical convolution neural network for ILSVRC classification task, the AlexNet. The size of the input image is $3 \times 32 \times 32$ while the batch size is 10, in other words, ten images are classified at the same time. Instead of using the average to represent the execution time, minimum execution time is selected, as the irrelevant factors only slow down the inference speed. The result is shown in figure **??**, which satisfies the expectations in the previous sections.

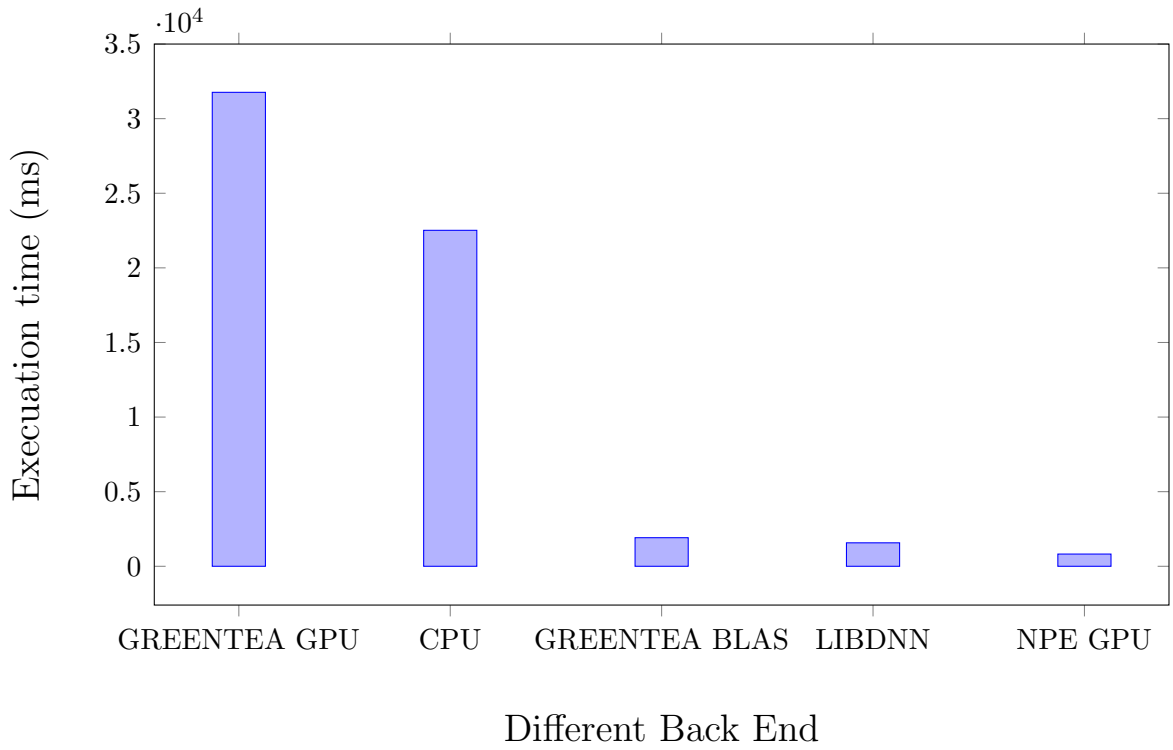


Figure 4.6: Performance of different Backends

Chapter 5

Convolutional Neural Network Compression: A GPU Version

In the past two years, ultra-large neural networks have dominated artificial intelligence area, with an impressive performance on lots of tasks, especially for image related tasks [8] [51]. However, the larger the neural network, the harder the deployment of these models, as they involve millions of parameters. Addition to storage overhead, computation cost is a more serious problem on mobile platforms.

On the other hand, the improvement of the performance is not proportional to the growth of the complexity. Canziani et al. [2] listed the relationship between the classification accuracy and number of operations, as shown in figure 5.1. It can be seen that, for ResNet-101 and ResNet-18, there is about 5% accuracy gain at the expense of nearly 3× more operations. However, it is worth to keep the neural networks deep during training, as the larger neural network, the larger functional space and the stronger learning abilities. This indicated that if ResNet-101 is trained first, and then compressed to its 40% of the original size, the classification accuracy should be

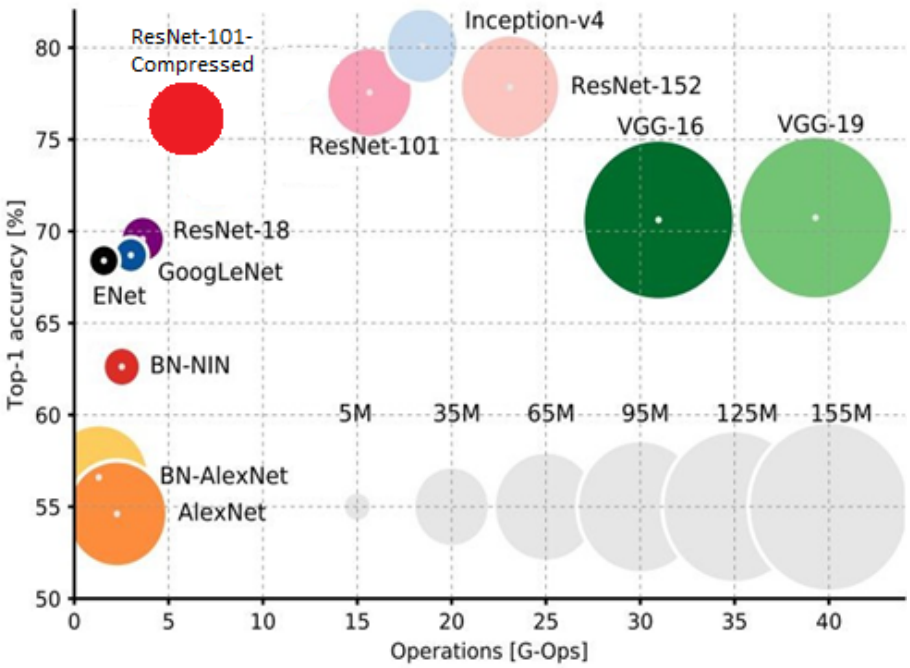


Figure 5.1: Relationship between the classification accuracy and number of operations [2]

better than ResNet-18 (see figure 5.1). According to the above assumption, we proposed convolution neural networks compression technique for mobile platforms.

5.1 Product Quantization Revisit

Several neural network compression techniques have been discussed in §III, which achieved more than 90% compression with less than 1% accuracy loss in fully connected layers. However, for convolutional layers, which takes more than 90% inference time, the compression methods used for fully connected layers cannot be used.

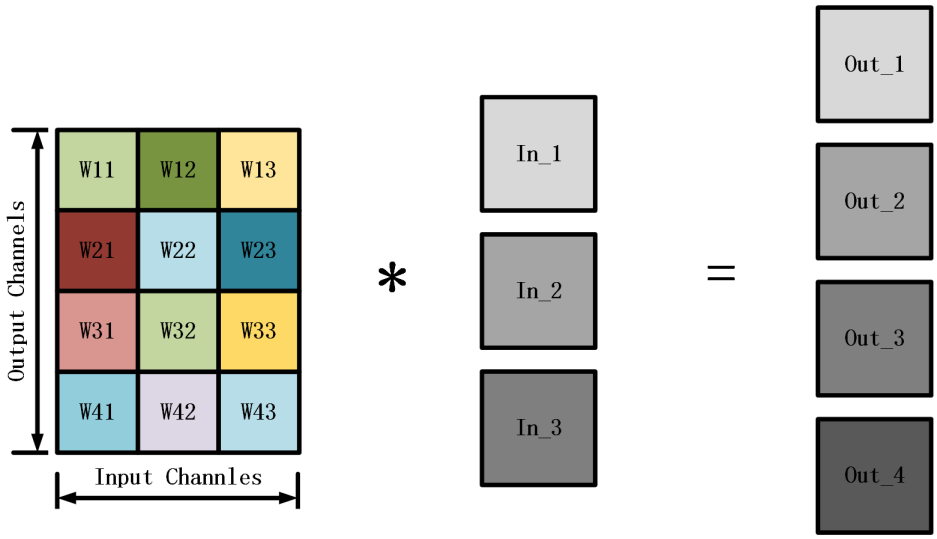


Figure 5.2: A brief illustration of convolution layers

The core operations of a convolution layer are sliding window convolutions. Convolution filters are stored as 4-dimensional tensors in CNNs, denoted as $W \in R^{K \times C \times R \times S}$, where K and C are the number of output channels and input channels, respectively, as shown is figure 5.2. For each $W_{ij} \in R^{R \times S}$, it is the convolution filter corresponding to the input channel i and the output channel j, where $R \times S$ is the filter size (see figure 5.3).

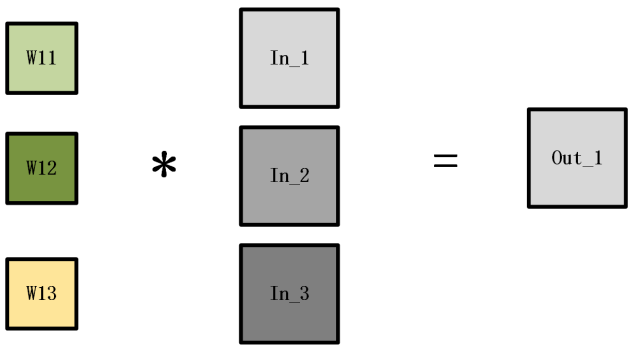


Figure 5.3: Convolution for same output channels

In the work that is most related to us, Gong et al [12]. proposed several similar compression algorithms for convolution neural networks, among

which, K-means clustering gives the least accuracy loss. The ideas of compression by K-means clustering is treating each convolution filter as a vector or a pointer in K-means clustering. As shown in figure 5.4, for this convolution layer, there are three input channels and four output channels. Convolution filters with similar colour means smaller distance between them. During K-means clustering, which filter is assigned to which cluster is recorded as code-book. However, they focused on storage reduction but not computation saving, in other words, even if the original neural network is compressed, before inference, the original neural network needs to be reconstructed.

Figure 5.4: Compression along input channels

The reason is shown in figure 5.5, in which W41 and W21 are compressed together, and represented by W2'1. However, as W41 and W21 are applied to different input channels, we still need to compute four convolutions instead of three ones. Even worse, as GPUs are SIMD (single instruction multiple data) computation units, compression in this way will cause memory divergence, which leads to poor performance. To solve this problem, in this chapter, we propose a algorithm that focuses on computation reduction and energy saving.

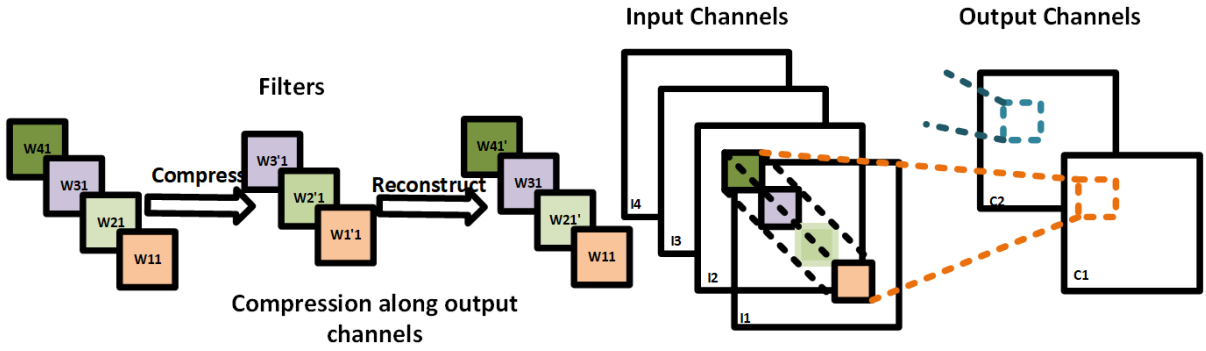


Figure 5.5: Compression along input channels

5.2 Channel-Oriented K-Means Clustering Compression

Overall Channels-Oriented K-Means Clustering Compression is conceptually a simple two-phase methodology: Channel-Oriented K-Means Clustering during the inference stage and converge points transition during the training stage. In this section, first, an efficient test-phase convolution method with network compression will be introduced. Secondly, a novel training strategy, named converge points transition will be proposed, by which better compression is achieved with fine-tuning of the entire network.

5.2.1 Channel-Oriented K-Means Clustering

In MobileDL, each tensor is divided into c sub-tensor groups, in which each sub-tensor is denoted as $W \in R^{K \times R \times S}$, and a group is mathematically defined as $G_m = \{W_{ij} | \forall j = m\}$. Each G_m is then be treated as a set of vectors $S = \{v_1, v_2, ..., v_k\}$, where each vector v_i is a $R \times S$ real vector reshaped from W_{im}. For each S, Channel-Oriented K-Means Clustering aims to partition these k vectors into $\tilde{k}$ sets by K-means clustering, then each set is represented

by one single vector $\tilde{v}_k$, and all representation vectors is denoted as a set $\tilde{S} = \{\tilde{v}_1, \tilde{v}_2, ..., \tilde{v}_{\tilde{k}}\}$. The whole process is shown in figure 5.6. The mapping matrix is denoted as M, where $M_{ij} = 1$ if v_j is assigned to $\tilde{v}_i$, otherwise $M_{ij} = 0$. Then S' is reconstructed from $\tilde{S}$ and M, mathematically, $S' = \tilde{S}M$, and the objective is to minimize "distance" between S and S'. Within-cluster sum of squares is chosen as loss function, mathematically, the follow objective function is going to be optimized:

$$min \sum_{i=1}^{n} \|v_i' - v_i\|^2$$

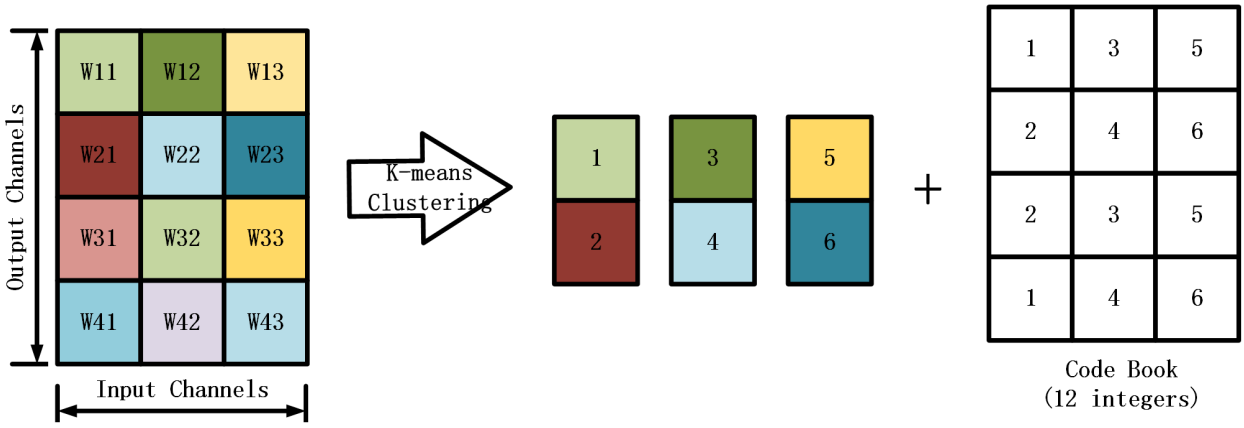


Figure 5.6: Convolution Filters Compression:

As the output dimensions are reduced from k to $\tilde{k}$, only $c \times \tilde{k}$ convolutions are actually computed, the temporary result is denoted as $O' = \left\{o_1', o_2', ..., o_{\tilde{k}}'\right\}$. Afterward, the original outputs will be approximately reconstructed from the temporary Output O' and the mapping matrix M, which is shown in figure 5.7 and mathematically expressed as

$$O = O'M$$

As a result, the overall time complexity will be reduced from $\mathcal{O}(C \times K \times$

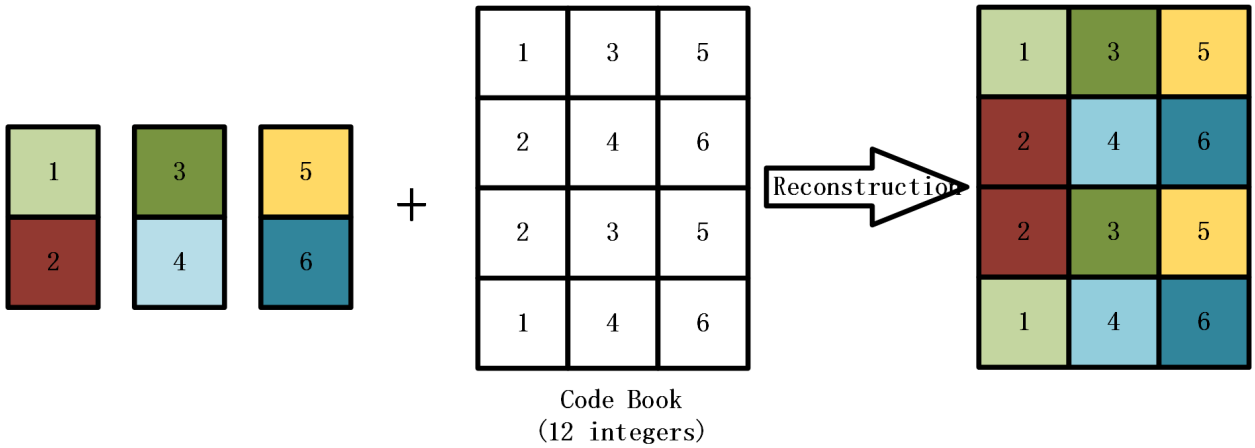


Figure 5.7: Convolution Filters Reconstruction:

$R \times S \times W \times H$) to $\mathcal{O}(C \times \tilde{K} \times R \times S \times W \times H + K \times W \times H)$. On the other hand, as only the clustered kernel and the mapping index need to be stored, the storage will also be reduced, as shown in figure 5.8.

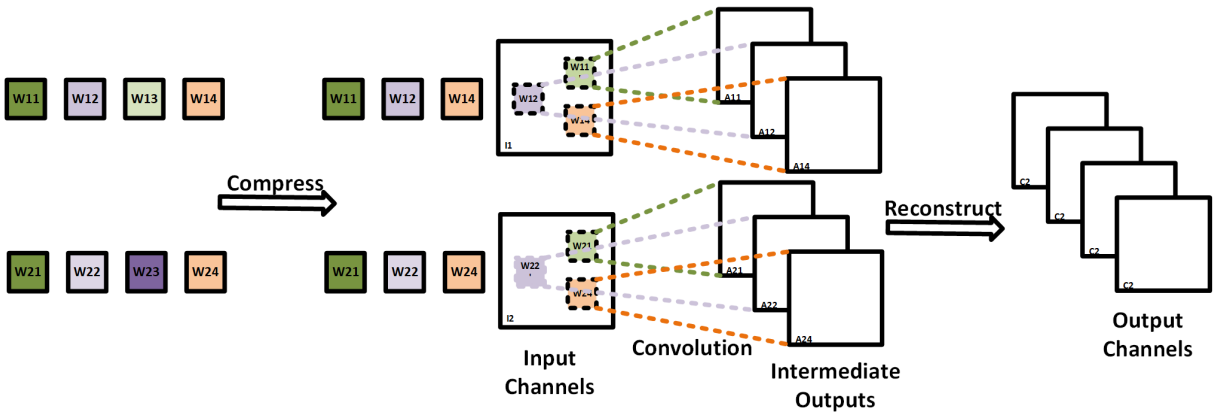


Figure 5.8: Compression along output channels

5.2.2 Converge Points Transition

With Channel-Oriented K-Means Clustering, the inference stage is significantly accelerated. However, there is still a critical drawback: the model which gives minimum before compression is not necessarily the one giving

minimum after compression. As there are numerous of acceptable minima of the objective function [28], a model which gives acceptable loss of the objective function before Channel-Oriented K-Means Clustering is usually not the one giving the best classification accuracy after Channel-Oriented K-Means Clustering. Furthermore, as Channel-Oriented K-Means Clustering is performed on each layer independently, the numerical error will be accumulated. The accumulated error may be intolerable if the network is deep. With CPT, the parameters of the model will be transited according to the objective function with least accuracy loss after Channel-Oriented K-Means Clustering.

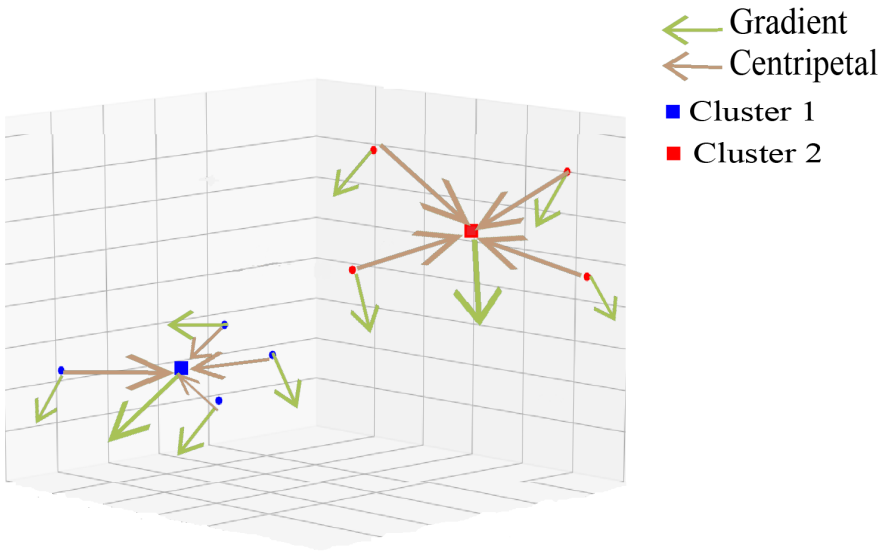


Figure 5.9: Converge Points Transition

CPT is essentially a modified gradient descent method that adds a centripetal descent factor to the original descent direction. For the state-of-the-art stochastic gradient descent, the parameters updating method is $\theta = \theta - \epsilon D$, where ϵ is the learning rate and D is the descent direction. Furthermore, the direction D is determined by two factor: gradient and regulariza-

tion. The updating function then can be expressed as $\theta = \theta - \epsilon(g + \alpha\nabla\Omega(\theta))$, where g is the gradient, α is the weight decay rate, and $\nabla\Omega(\theta)$ is the regularization factor, for example, if L^2 regularization is used, then $\Omega(\theta) = \frac{1}{2}||\omega||_2^2$. This regularization strategy drives the weights closer to the origin, while CPT approach is based on driving the weights close to each other within same cluster to limit the variance of model. The CPT factor is calculated as

$$\Psi(\theta) = reshape\left\{S_1\tilde{M}_1, S_2\tilde{M}_2, ..., S_c\tilde{M}_c\right\} - \theta$$

The new updating function now is

$$\theta = \theta - \epsilon(g + \alpha\nabla\Omega(\theta) + \frac{\beta}{\epsilon}\Psi(\theta))$$

This method is intuitively illustrated in Figure 5.9. For simplification, each high-dimension vector is expressed as a point. Vectors assigned to the same cluster are in the same color. A cluster is represented by the mean of vectors within it and is expressed as a small square. To further simplify the problem, regularization factor is ignored. The descent direction of a vector can be treated as the combination of gradient direction and centripetal direction. If a cluster is regarded as a system, then the CPT factor can be regarded as gravitation in a galaxy, which points to the center of the galaxy. As the summation of gravitation between every two stars is zero in a closed galaxy, all CPT factor in a closed parameter space will also be summed to zero, in other words, the centripetal descent will cancel each other, mathematically,

$$\Sigma_i^k(\tilde{s}_i - v) = \Sigma_i^k\tilde{s}_i - kv = kv - kv = 0$$

Hence, only the gradient descents contribute to resultant descent, which applied to the mean-vector, and the resultant descent is expressed as $D_s = G_s =$

$\Sigma_i^k g_i$. Objective function will descent along the gradient in the granularity of cluster. Hence, after adding the CPT factor, from system point of view, it is still the same optimization problem as before. Because within-cluster descent and between-cluster descent are independent, the whole parameter will also converge.

5.2.3 Pseudo-code

The Pseudo-code of Channels-Oriented K-Means Clustering Compression is shown in **Algorithm** 1. There are two hyper parameters need to be chosen manually. The first one is *class_num*, which determines the compression rate (i.e. compression rate = $\frac{class\_num}{filter\_num}$). According to experiments, the compression rate should be about 50%, in order to balance between accuracy loss and speed up. In other words, the *class_num* should be chosen as $\frac{filter\_num}{2}$. However, different with *class_num*, the best choice of second parameter *eps* varies among different neural networks. There is only empirical suggestion for the choice of *eps*: for complicated neural networks, the esp should be around 0.1; while for smaller neural networks, *eps* could be around 0.25.

Algorithm 1

```
def K-Means_Clustering_Compression(model, data):
    for i in range(iteration):
        loss = compressed_forward(model, data)
        CPT_direction = SGD_with_K-Means(model, eps, class_num)
        model.parameters = model.parameters - loss * (1-eps) +
            esp * CPT_direction

def SGD_with_K-Means(model, loss, class_num):

    groups = dict{}

    for filter in model.filters:
        Groups[filter.in].append(filter)
```

```
    for g in groups:
        centers, code_book = K_Means_Clustering(g, class_num)
        for filter in g:
            new_pos = centers[code-book[[filter.in][filter.out]]]
            CPT_direction[filter.in][filter.out] =
                g[filter.in][filter.out] - new_pos

    return CPT_direction

def compressed_forward(model, data):

    temp_result = convolution(model.compress_filter, data)

    for t in range(model.filter.out):
        for pos in range(code-book_inverted[filter.out]):
            result[t] += temp_result[code-book_inverted[filter.out][pos]]

    return result
```

5.3 Mask-Based Neural Network Recovery Compression

In this section, firstly, we will introduce a novel neural network compression method named Neural Network Recovery Compression. Different with K-Means Clustering Compression, which focuses on convolution neural network, Neural Network Recovery Compression method works on any kinds of deep neural networks. Then, an improved version of this algorithm, name Mask-Based Neural Network Compression, which are much faster and robuster, will be discussed in detail.

5.3.1 Neural Network Recovery Compression: A Simple Version

Neural Network Recovery Compression method is inspired by the recovery of human brains. Some researches indicate that if some parts of human brains is injured, some other parts will take over the functions of those parts gradually. Meanwhile, Denil et al. also demonstrated that there are huge

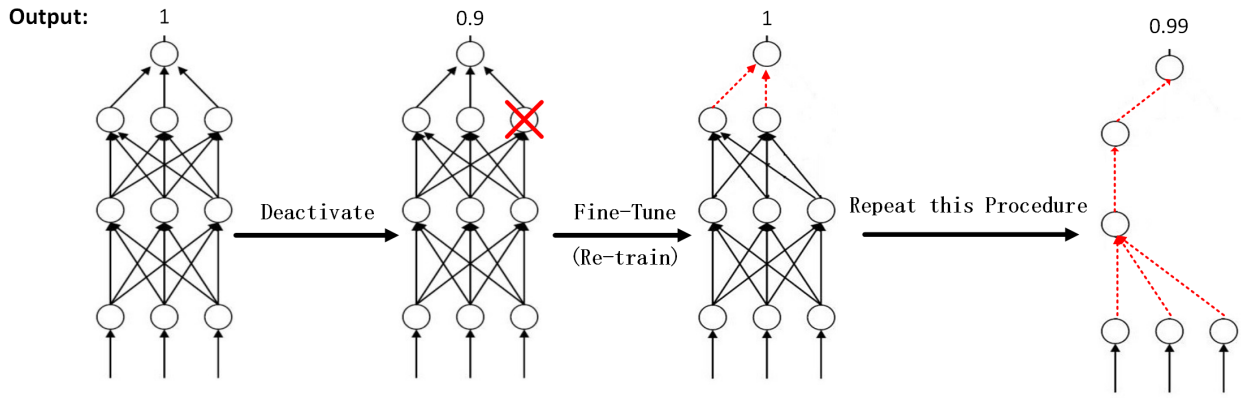


Figure 5.10: Neural Network Recovery Compression

redundancies in neural networks [23]. Therefore, if some parts of the whole neural network is deactivated manually ("injured"), the whole neural network is able to recovery to the original performance after several iteration of retraining("brains recoveries"). Han has proposed a algorithm about connection learning [52], the whole procedure is shown in figure 5.10. For each iteration, they select a small number of nodes of the whole parameter space. Then, this small subset is deactivated, which means that all weights in these nodes are set to 0. Finally, the whole neural network needs to be retrained to cancel the influence of deleted parameters. The whole procedure needs to be repeated until the compression rate is achieved, in other words, there are enough nodes deleted. The detailed description of the most straightforwards implementation is shown in **Algorithm** 2.

Algorithm 2 (Song's method)

```
def Neural_Network_Recovery_Compression(model):
  while the accuracy_loss is acceptable:
    accuracyd_befored_injuries = eval(model)
    select a small subset of the whole parameter space
    setting all parameters in injured to 0
    while there is performance increment:
      re-trained the whole neural network
      accuracyd_afterd_injuries = eval(model)
      accuracyd_loss = accuracyd_befored_injuries
        - accuracyd_afterd_injuries
```

5.3.2 Neural Network Recovery Compression with Mask

Even though the algorithm proposed in the previous subsection runs in O(n), where n is the number of parameters. However, there are huge amounts of parameters in state-of-the-art deep neural networks. According to Culurciello's study, for ResNet-50, there are about 3 million parameters in total(see figure 5.1). Therefore, even for 50% compression ratio, about 20 million iterations are needed, which may takes several days on typical workstations.

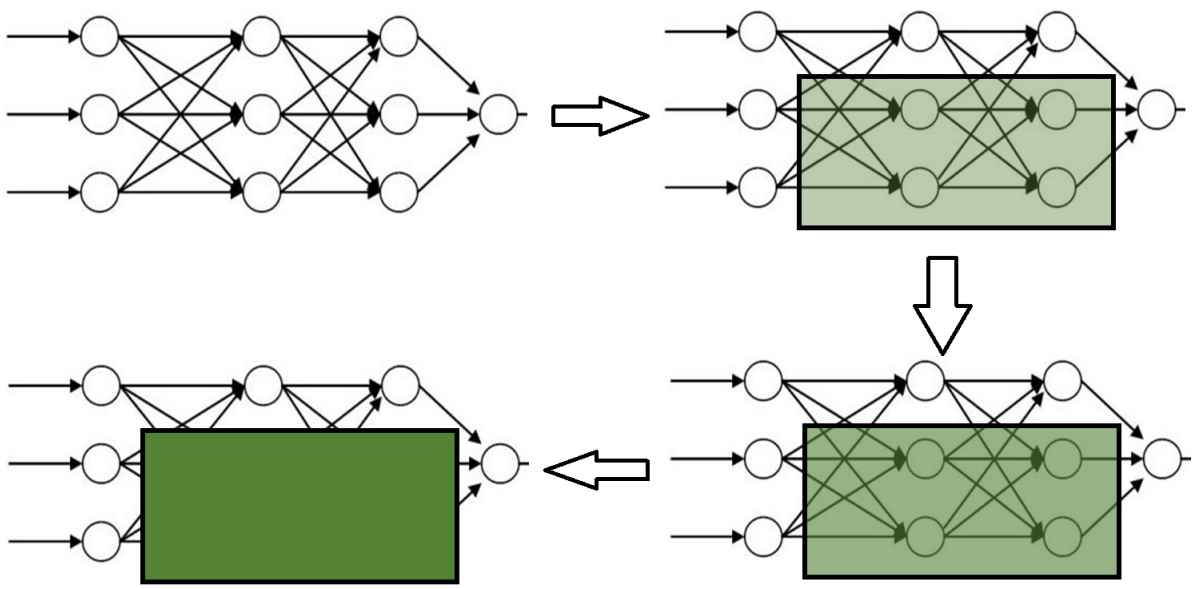


Figure 5.11: Neural Network Recovery Compression: The Mask Implementation

In order to solve the curse of parameter numbers, another version of neural networks recovery compression, named Mask-Based Neural Network Compression is proposed. Instead of fully deactivate a small subset of the whole parameter space (setting all of them to 0), a deactivation mask is application to the whole portion of the parameter space, that is going to be pruned

gradually. The high level ideas of the algorithm is shown in figure 5.11. The progress of the neural network recovery compression is same as the training, except there is a deactivation mask involved. The deactivation mask is like a gate that controls the activation of parameters. At the beginning, the value of the deactivation mask is set to 1, which means that all parameters can remain their values. With the progress of the compression, the value of the deactivation mask is drop from 1(all pass) to 0(no pass). Finally, no parameters can pass the deactivation mask, in other words, the portion applied with the mask is non-functioned. Therefore, the whole portion can be deleted from the neural network. The formal description of the algorithm is shown in **Algorithm** 3.

Algorithm 3

```
def Mask-Based_Neural_Network_Recovery_Compression(model, benckmark_model):
    mask_pos = []
    param_num = num(model.parameters)
    for i in range(1, compression_rate * param_num):
        pos = model.parameters[random_int]
        mask_pos.append(pos)
    #randomly selecting a large part of the whole parameter space
    for i in range(compression_iterations):
        mask = 1 - i / compression_iterations
        #can be other function
        for p in mask_pos:
            p = p * mask
        for i in range(1, itr):
            loss = forward(model, data)
            loss = loss / mask #keep learning rate
            model.update(loss)
        #re-train the whole network for several iterations

    benchmark_acc = benchmark_model.evaluation(test_data)
    compressed_acc = model.evaluation(test_data)

    if (benchmark_acc - compressed_acc > thredhold):
        return model
```

Chapter 6

Experiments and Results of Compression

6.1 LeNet on MNIST dataset

The MNIST dataset was used to evaluate the performance of different compression methods for LeNet. The MNIST database is the most famous hand-written digits dataset that contains 60,000 training images and 10,000 testing images.

LeNet, which is proposed by LeCun in 1998, is the most famous neural network that used for hand-written digits recognition. There are two convolution layers, and two fully connected layers, the detailed architecture of LeNet is shown in Table 6.1.

6.2 AlexNet on Cifar-10 dataset

The CIFAR-10 dataset consists of 60000 color images in 10 classes, such as airplanes, automobiles, birds, cats, etc. There are 6000 images per class.

AlexNet is not the state-of-the-art neural network for object recognition.

Number	Type	C	K	$W_1 \times H_1$	$W_2 \times H_2$	$R \times S$
1	CONVOLUTION	1	20	32×32	32×32	5×5
2	MAX POOLING	20	20	32×32	16×16	N/A
3	CONVOLUTION	20	50	16×16	16×16	5×5
4	MAX POOLING	50	50	16×16	16×16	N/A
5	FULLY CONNECTED	1	1	$50 \times 16 \times 16$	500×1	N/A
6	RELU	1	1	500×1	500×1	N/A
7	FULLY CONNECTED	1	1	500×1	10×1	N/A

Table 6.1: Architecture of LeNet [5]

Number	Type	C	K	W_1×H_1$	W_2×H_2$	$R \times S$
1	CONVOLUTION	3	96	227×227	55×55	11×11
2	MAX POOLINT	96	96	55×55	55×55	N/A
3	CONVOLUTION	96	256	55×55	27×27	5×5
4	MAX POOLING	256	256	27×27	27×27	N/A
5	CONVOLUTION	256	384	27×27	13×13	3×3
6	CONVOLUTION	384	384	13×13	13×13	3×3
7	CONVOLUTION	384	256	13×13	13×13	3×3
8	MAX POOLING	256	256	13×13	13×13	N/A
9	FULLY CONNECTED	1	1	43264×1	4096×1	N/A
10	FULLY CONNECTED	1	1	4096×1	4096×1	N/A
11	FULLY CONNECTED	1	1	4096×1	10×1	N/A

Table 6.2: Architecture of AlexNet

However, the size of the AlexNet is believed to be the same with a typical neural network that executed on mobile devices. The detailed architecture of AlexNet is shown in Table 6.2, in which there are five convolution layers and five fully connected layers.

6.3 Equipment

In this project, the focus is on optimizing the inference stage on mobile GPUs. For training, powerful workstations are still needed. The configu-

ration of the workstation used in this project is shown in Table 6.3, which is affordable for most small companies or even independent developers. For the inference stage, MobileDL can be executed on any devices that support OpenCL. However, as MobileDL is optimized for mobile devices with GPUs, to make best of MobileDL, mobile devices with at least 2 GB RAM and 16 GB storage are preferred. The platform used for test in this report is the Snapdragon 835 development board.

CPU	Intel Xeon E5-2630	$\times$ 2
GPU	GeForce GTX 1080	$\times$ 2
Memory	64GB DDR4	
Disk	Samsung SSD 850	

Table 6.3: Configuration of training workstation

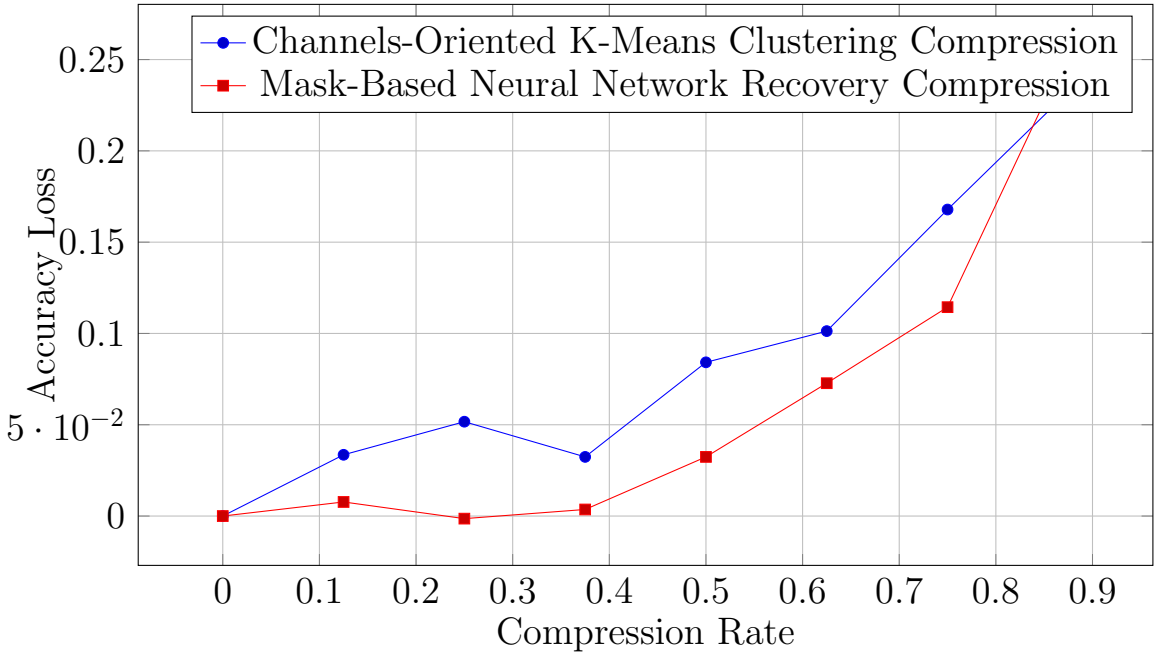
6.4 Experiments

The experiments of both Channels-Oriented K-Means Clustering and Mask-Based Neural Network Recovery are performed on the tests described in previous sections. For K-Means Clustering, neural networks mentioned in previous sections were pre-trained with different compression rates with CPT. Then K-Means Clustering was used to compress the original neural network. While for Mask-Based Neural Network Recovery Compression, the same pre-trained models are also used for the experiments.

6.4.1 Classification Error

Figure 6.1 (AlexNet) and figure 6.2 (LeNet) show the classification error of Channels-Oriented K-Means Clustering Compression and Mask-Based Neu-

Figure 6.1: Classification Error of AlexNet

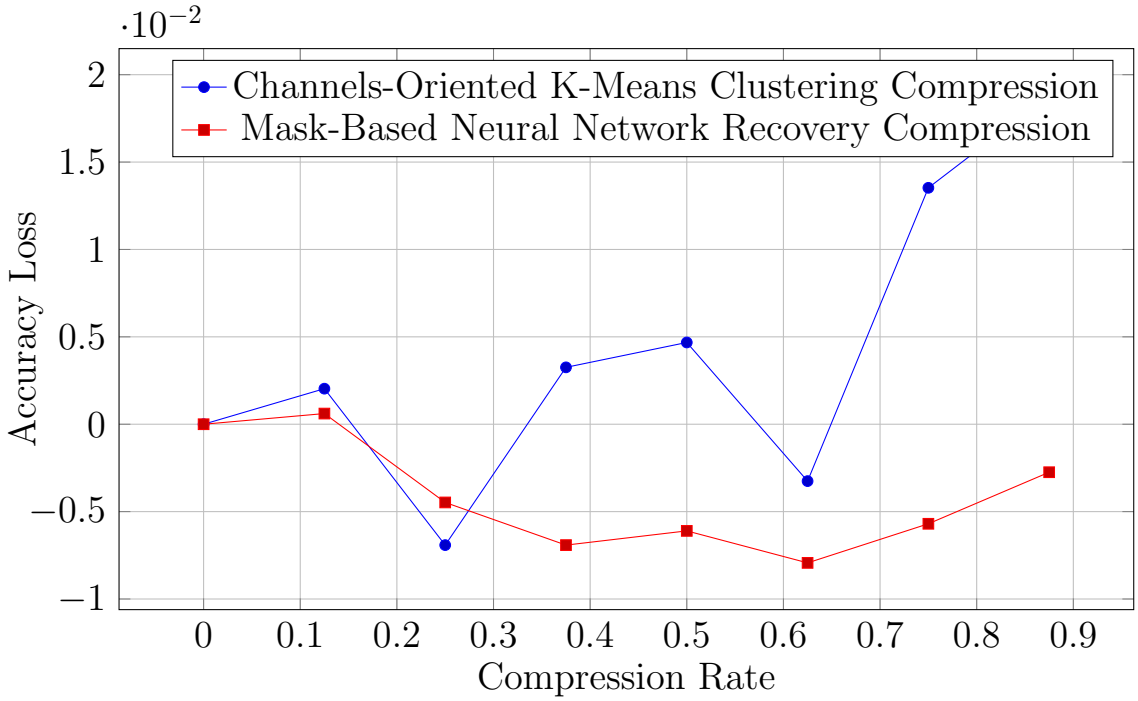


ral Network Recovery with different compression rates. The neural network without compression is chosen as the baseline.

It can be seen from figure 6.2 that for LeNet on MNIST, both compression methods performs extremely well. Even with 87.5% compression ratio, the accuracy loss is mere -0.0027(Mask-Based Neural Network Recovery) and 0.0188(Channels-Oriented K-Means Clustering). The reason is that: (1) digit recognition task is an extremely easy task, and (2) there is so few features need to be extracted, which means that a great portion of parameters is redundant.

while for AlexNet on Cifar-10, the difference between these two compression methods is more significant. For Channels-Oriented K-Means Clustering, a compression rate of 50% will cause about 8.5% classification error (see Figure **??**), in other words, statistically, for 100 mis-classifications of the uncompressed neural network, there will be 108 images be classified incorrectly

Figure 6.2: Classification Error of LeNet



by the compressed neural network. The classification error is tolerable, especially for mobile application, which is less critical than professional software.

On the other hand, Mask-Based Neural Network Recovery Compression performs better than Channels-Oriented K-Means Clustering on both AlexNet and LeNet. It can be seen from the figure 100 that, even for a compression ratio of 67.5%, the accuracy loss is mere 7.2%. Meanwhile, Mask-Based Neural Network Recovery Compression is more robust than Channels-Oriented K-Means Clustering, in other words, it is easier to train deep learning model with Mask-Based Neural Network Recovery Compression. However, some experiments performed on larger neural networks, which is not supposed to be executed on mobile devices in current stage(but may not be true in the future), indicated that Mask-Based K-Means Clustering performs better than

Channels-Oriented Mask-Based Neural Network Recovery Compression.

The result is much better than our expectation, the reason may be that for a neural network with n layers, where each layer has d parameters, its functional space has d^n local minima, and for a high order function, every minimum will give an acceptable loss. As there are d^n possible minima, it is more likely to find the local minimum that is suitable for our clustering method.

6.4.2 Speed Up

Figure 6.3 and figure 6.4 show the empirical speedup we achieved on the Snapdragon 835 platform with Channels-Oriented K-Means Clustering Compression and Mask-Based Neural Network Recovery Compression. For Mask-Based Neural Network Recovery Compression, 2.28× speedup is achieved with 67.5% compression. Recent years, with the development of mobile processors, the performance gap between desktop CPUs and GPUs have been shortened that some state-of-the-art mobile processors (e.g. Snapdragon 835) are as powerful as desktop CPUs (e.g. Intel 7th i5). Therefore, 2.28 times speedup makes our deep learning frameworks for mobile platforms even faster than some desktop ones.

For Channels-Oriented K-Means Clustering, theoretically, if the compression rate is 50%, as there is only half of computation needed, about 2× speedup should be achieved. However, it can be seen from figure 6.3 that for Channels-Oriented K-Means Clustering compression, with 50% compression, the speedup is 1.31×, which is much less than 2×. The reason for

Figure 6.3: Speed up of AlexNet

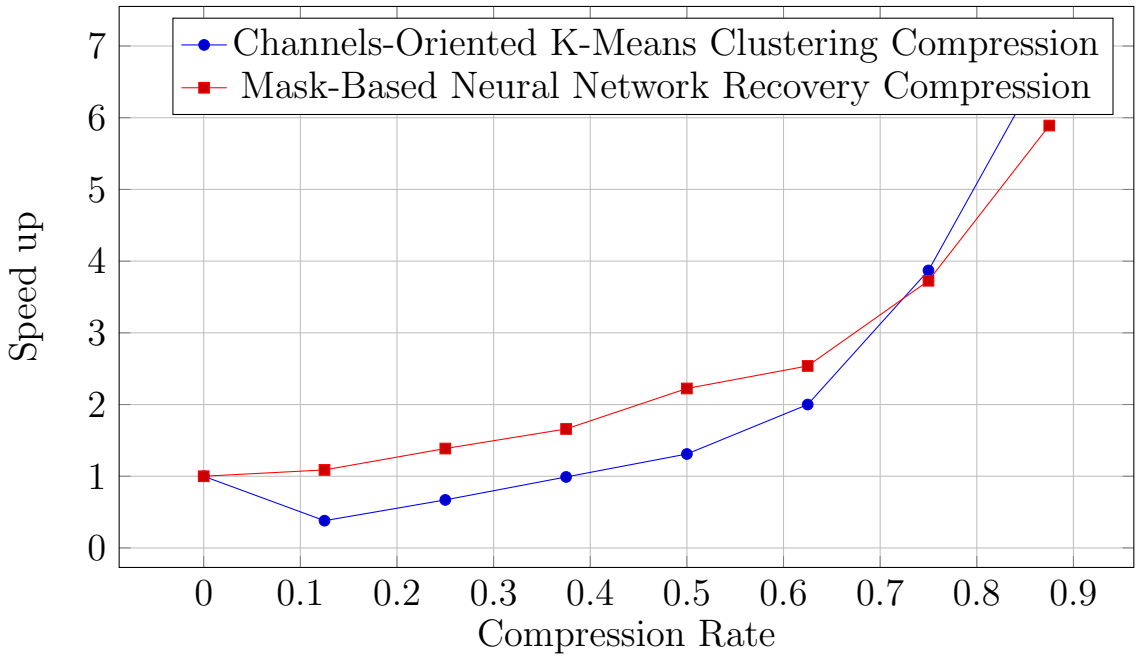
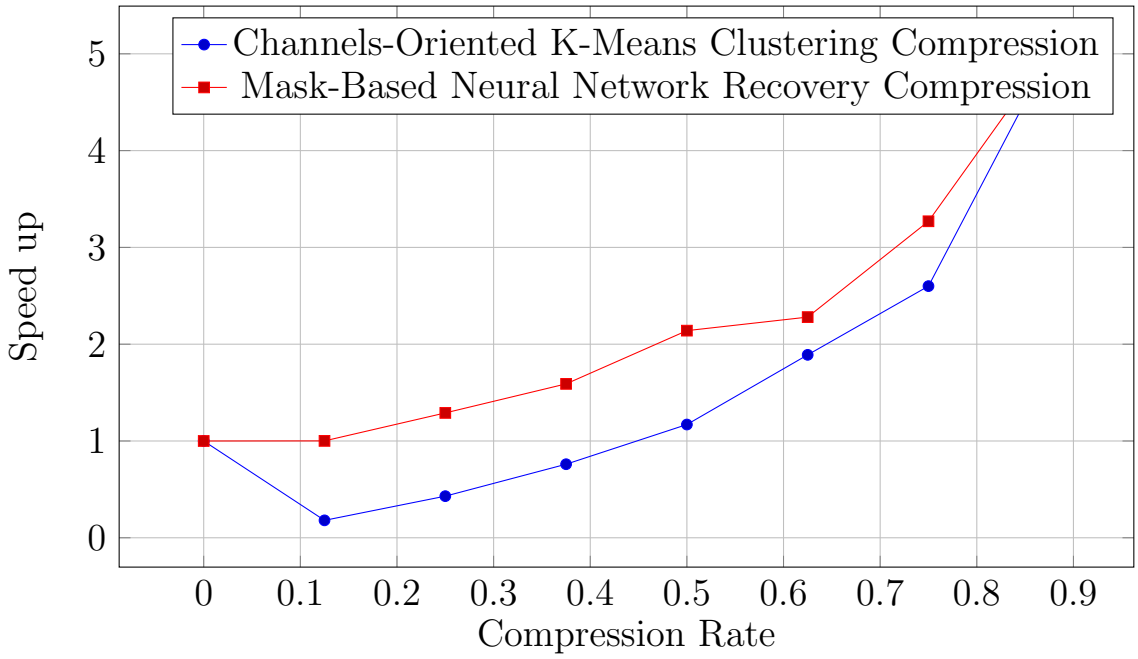


Figure 6.4: Speed up of LeNet



the gap between theoretical and actual speed is that even though the convolution computation is reduced from $\mathcal{O}(C \times K \times R \times S \times W \times H)$ to $\mathcal{O}(C \times \tilde{K} \times R \times S \times W \times H + K \times W \times H)$, as explained in the algorithm section, there is still $\mathcal{O}(K \times W \times H)$ rather than $\mathcal{O}(\tilde{K} \times W \times H)$ memory access. Meanwhile, as illustrated in figure 5.8, even if the upper bound of memory access remains unchanged, there are $\mathcal{O}(\tilde{K} \times W \times H)$ more memory access during reconstruction, which also hurts the overall performance. For a deep neural network with C input channels and K output channels, if the compression rate is r, then the memory overhead is $\frac{r \times \tilde{K} \times W \times H}{C \times K \times W \times H}$, which is around 10% in this experiment. Therefore, even if we trade about 10% memory overhead for 50% computation work, we can only get approximately 1.31× speedup, as the memory access is much more expensive than computation on integrated memory architecture.

However, it is not indicated that Neural Network Recovery Compression is better than Channels-Oriented K-Means Clustering Compression. It can be seen from figure 100 that, no matter for speedup or accuracy loss, Channels-Oriented K-Means Clustering performs better than Neural Network Compression. In other words, Channels-Oriented K-Means Clustering performs better on ultra large neural networks. Even though ultra large neural networks are too large to be executed on mobile platforms in the current stage, it is believed that mobile processors can hold such large neural network in the near future, and Channels-Oriented K-Means Clustering Compression works well for those neural networks.

Chapter 7

Half Precision: A software implementation of NVIDIA Volta Tensor Core

To achieve higher accuracy, the complexity of deep neural network architecture has been increasing, which in turn lead to the growth of computation work. Mixed-precision training, proposed by NVIDIA [6], is a potential solution as it lowers the required resources. Compared to 32 bits for float, half, which only uses 16 bits, significantly decreases the required amount of memory. Meanwhile, with hardware support, half precision arithmetic offers 2× speedup compared to single precision. Additionally, for some memory sensitive task, more speedup is achieved due to the reduction of the number of bytes access, for example, NVIDIA TITAN V offers 8x more half precision arithmetic throughput in some extreme cases [53].

In this chapter, a brief explanation of how NVIDIA achieves Mixed-Precision is given. After that, a software solution that imitates NVIDIA ideas is proposed. Finally, NVIDIA's experiments of accuracy compare between float and half are listed.

7.1 Overview of Half Precision Supporting in NVIDIA Volta

7.1.1 Float16: half

In IEEE 754 format, float16 consists of 1 sign bit, 5 exponent bits, and 10 fractional bits (see figure 7.1). As there is only 16 bits rather then 32 bits, the range of float16 is much narrower than float. The range of positive normal range is $[6.10352 \times 10^{-5}, 65504]$, while the range of positive subnormal range is $[5.96 \times 10^{-8}, 6.10 \times 10^{-5}]$. Except that single precision is the lowest arithmetic precision for almost all current processors, another important reason of most deep neural networks are trained with single precision is that the range of half may not enough in some cases. For example, for convolution operations, the output of each convolution between each input channel and convolution filter is accumulated together, which may go beyond the range, especially when the number of the channels is large. To solve this problem, NVIDIA proposed a training method named mixed precision training [6], which multiplies half precision matrices and accumulate the result into either single- or half-precision output.

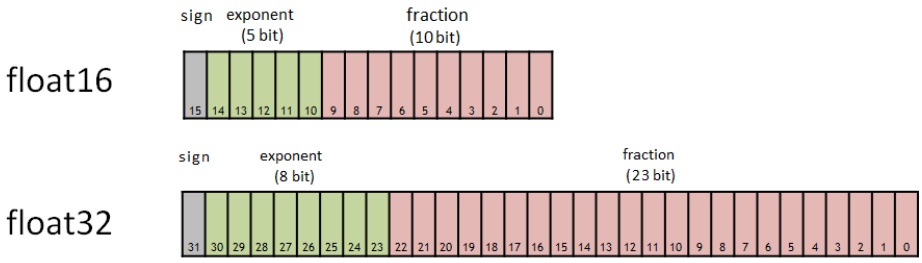


Figure 7.1: Bits usage of Float16 and Float32 in IEEE754

7.1.2 Mixed Precision Training

There are mainly three kinds of training methods in respect of precision supported by NVIDIA's libraries, which are listed in table 7.1.

Training Values Storage	MM Accumulator	Name
Float 32	Float 32	Float 32 Training
Float 16	Float 32	Mixed Precision Training
Float 16	Float 16	Float 16 Training

Table 7.1: Different Training Strategy

With 16 half-precision training values storage and single precision matrix-multiplication accumulator, mixed precision training with single precision master weight storage is proved to be the most suitable combination for deep neural network training. The basic idea of mixed precision training is illustrated in figure 7.2. For calculation of $D = AB+C$, the matrix multiplication of AB is computed under half precision, while during accumulation, it is under single precision.

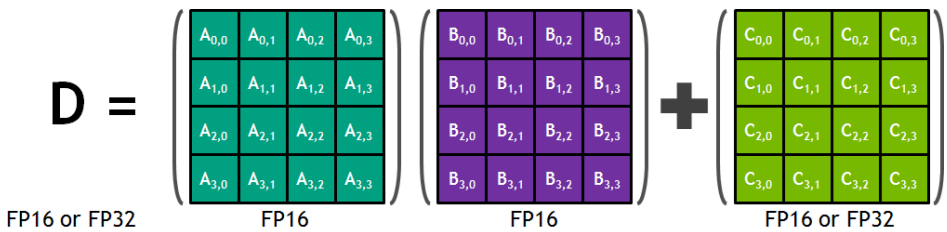


Figure 7.2: Convolution Computation by TensorOps:

In NVIDIA Volta architecture, Tensor Core Instructions is introduced, which multiply half precision matrices and accumulate the result in single or half precision natively [54]. For example, in convolution layers, convolution filters are applied to different input channels and then accumulated to the

output channels. In NVIDIA Volta architecture, these operations are illustrated in figure 7.3. It should be emphasized that there is only one conversion from half precision to single precision is performed after all computations.

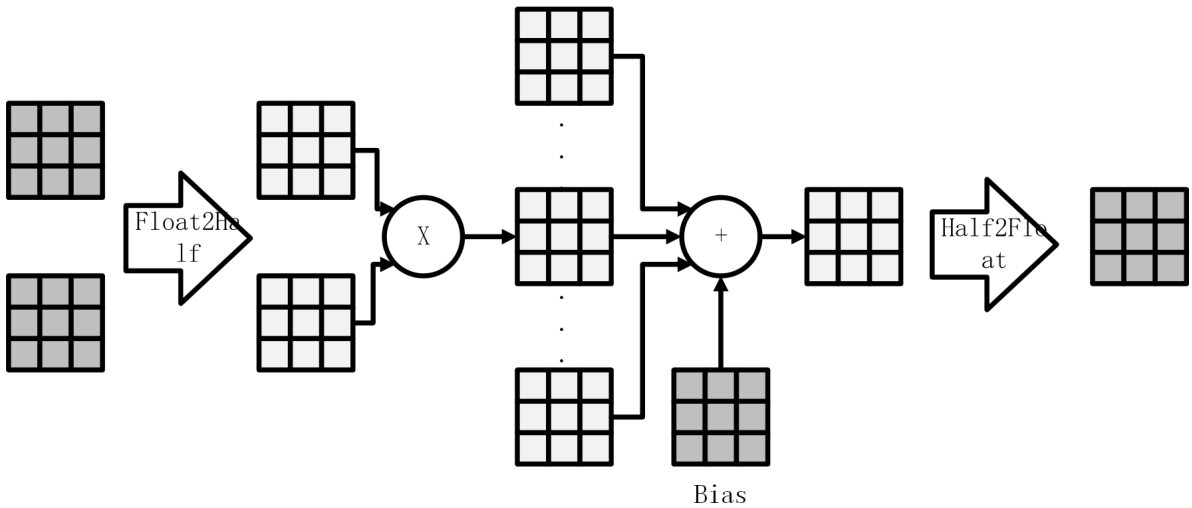


Figure 7.3: Brief illustration of NVIDIA's solutions [3]

7.2 Software Implementation of Half Precision

However, to benefit from this technologies, some hardware features that only available on NVIDIA Volta architecture with CUDA are needed, while for mobile platforms, most processors only support OpenCL. Even worse, due to NVIDIA's business plan, it is less likely that they will make these libraries open source, which means that it is difficult to improve or implement new algorithms based on it.

To solve this problem, a software implementation of NVIDIA's mixed precision training method is proposed in this final year project. To be honest, except the native support of accumulation of different precision matrics, all feature needed to take advantage of half-precision has already been available on Adreno GPUs. To imitate the tensor core, all half-precision matrices are

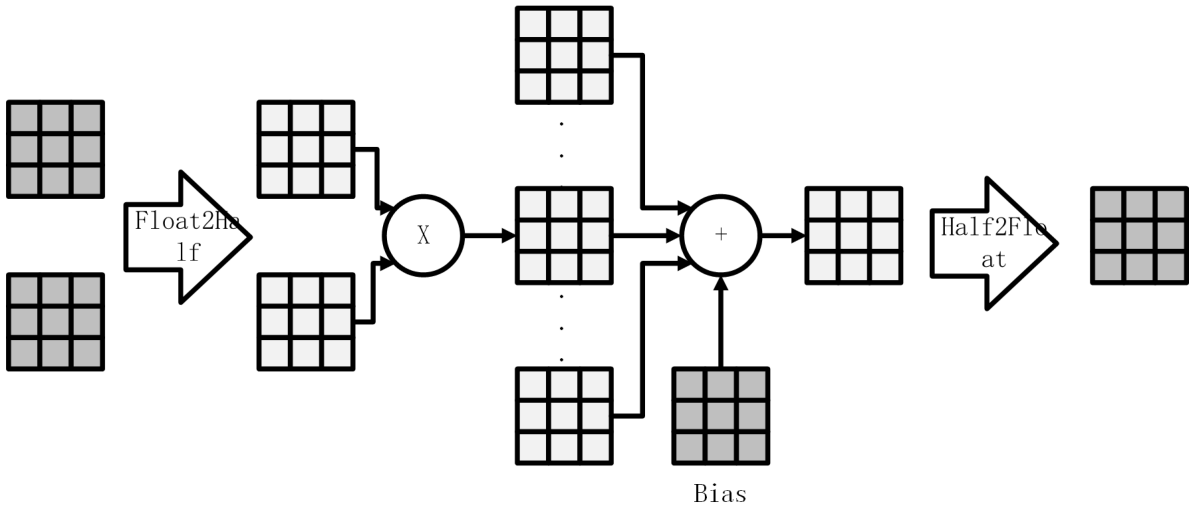


Figure 7.4: Brief illustration of our software solutions

converted to single-precision before accumulation explicitly. The high-level idea is illustrated in figure 7.4, from which it can be seen that n times conversion is needed for the software solution rather than one conversion for the native one. Even worse, the conversion should be performed by CPU. If the number of channels is large, the overhead is not ignorable. Fortunately, with the help of Zero-Copy technique discussed in the next chapter, the overhead will not suppress the benefits.

7.3 Experiment Result

There is no difference between NVIDIA's solution and our software implementation, except speed. That is, for the same setup, our solution will give the same result as NVIDIA's, with longer execution time. Therefore, in this report, NVIDIA's results are used, in other words, all results related to accuracy is referred from NVIDIA's experiments as training so many models on mobile devices are almost impossible. They trained several convolution neural networks for ILSVRC classification task with mixed precision train-

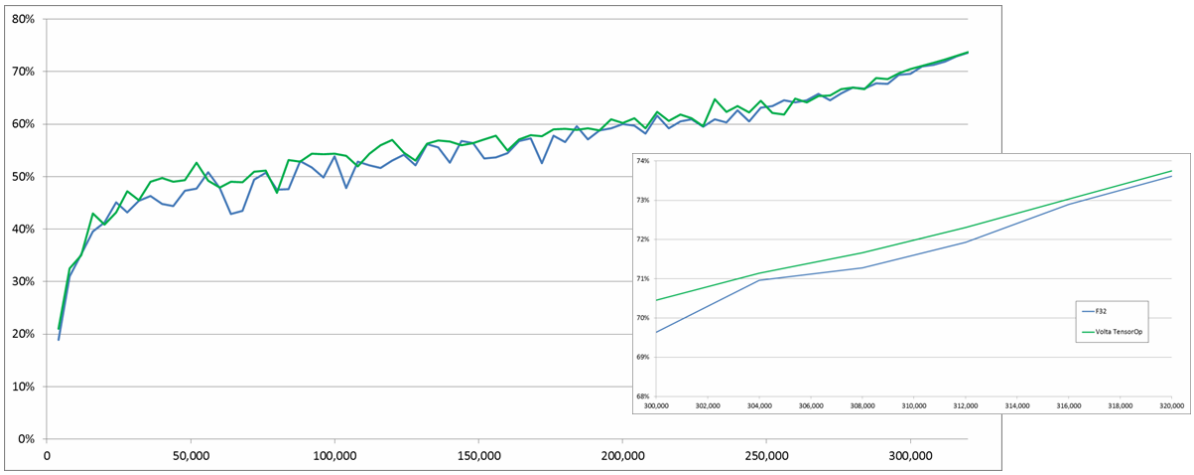


Figure 7.5: Accuracy figure of Resnet50 training under float16 and float32 [3]

ing method [6], which includes GoogleNet, inception v1 and Resnet50. They also used Caffe with modification with TensorOps to train these models and achieve the same accuracy with float-precision baseline using same hyper-parameters. The classification accuracy was reported in Table 7.2, and for

Model	Mixed Precision	Baseline
GoogleNet	68.43%	68.33%
Inception v1	70.02%	70.03%
Resnet50	73.75%	73.61%

Table 7.2: Comparison on various of GPUs [6]

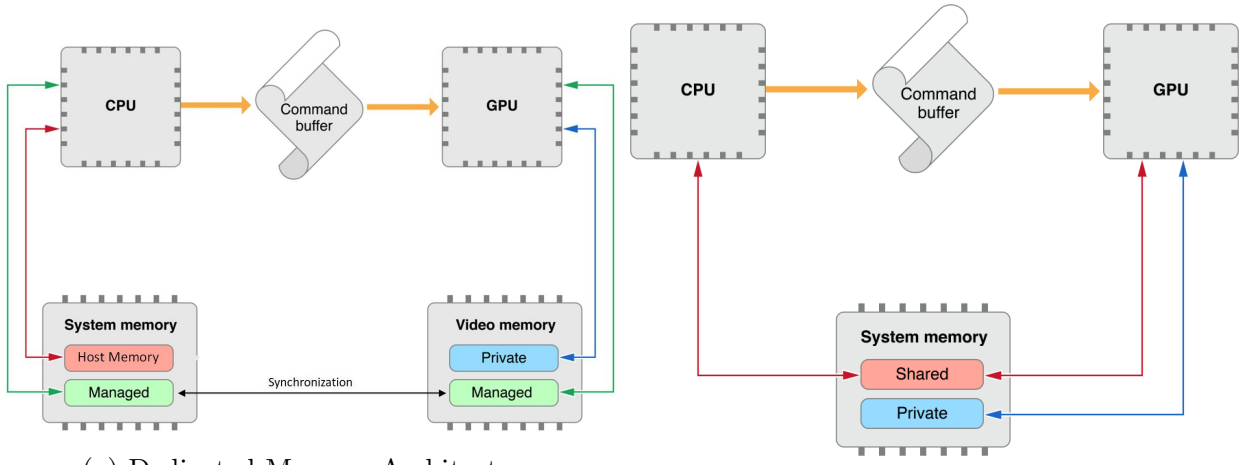
Resnet50, the accuracy figure was also provided in figure 7.5. They demonstrated that lots of deep neural network could be trained using mixed precision training with only minor accuracy loss even without hyper-parameter tuning.

Chapter 8

Unified Memory: Zero Copy between GPUs and CPUs

Memory copy is extremely expensive and energy-consuming. According to the standards specification of Low Power DDR (LPDDR) [32], the power consumption is 40 pJ/bit. However, for deep learning applications, for each layer, hundreds MB of data will be copied, even worse, most of the mobile deep learning applications run on battery power devices, in which reducing power consumption has higher priority than speed up.

To avoid costly memory copy, a mechanism, so-called Unified Memory, is adopted by Adreno GPUs [1]. In this chapter, memory architectures of dedicated and integrated GPUs are compared. Then, two memory management protocols between CPU-access memory and GPU-access memory are discussed. After that, the implementation details are introduced. Finally, the experiment result is provided.



(a) Dedicated Memory Architecture

(b) Integrated Memory Architecture

Figure 8.1: Memory Architecture of Desktop Platforms and Mobile Platforms [4]

8.1 Memory Architectures

Device memory models vary by different platforms. Desktop platforms support discrete memory model or dedicate memory model, while for mobile platforms usually shared memory model, in which the CPU and the GPU share system memory, is support. The differences are shown in figure 8.1.

There are mainly two differences between these two different memory architectures. The first one is that, for video memory that dedicated to "video" usage, it is 10× faster than system memory. This indicates that if video memory is available, it is worth to take the cost of extra copy and explicit synchronization so that further memory accesses are much faster. Another one is that for system memory, a memory block can be declared as shared, that is, this memory block can be accessed by both GPUs and CPUs. In this architecture, the shared memory does not need to be managed explicitly, no

matter the copy or the synchronization are handled automatically. Furthermore, if unified memory is supported by hardware, there will be no real copy and synchronization.

8.2 Memory Management Protocols

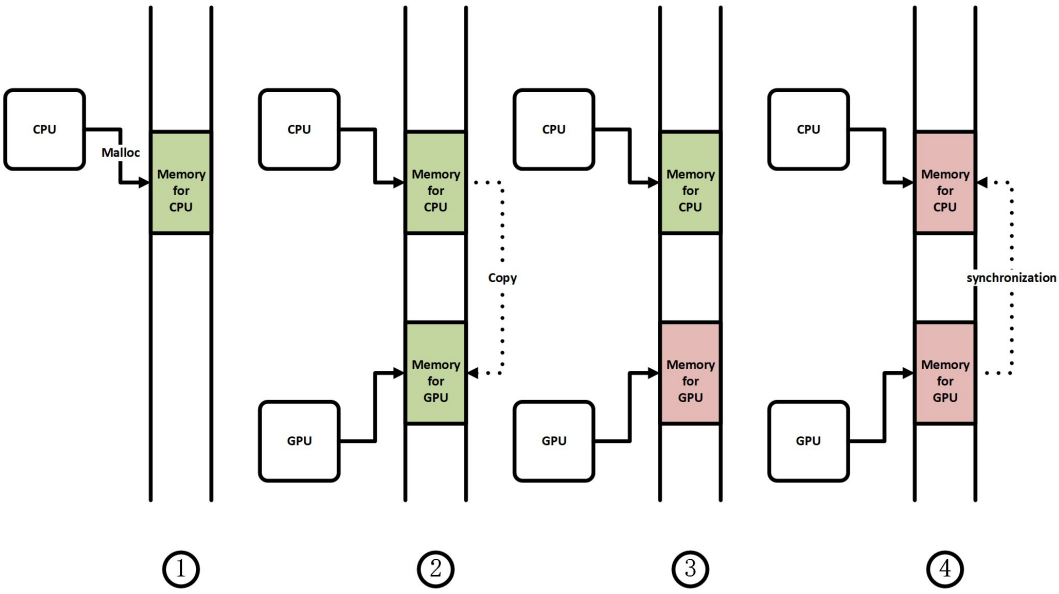


Figure 8.2: Private Memory on Integrated Memory Architecture

As shown in figure 8.1b, even for memory architecture of mobile platforms, memory blocks can be declared as shared or private. It is no good or bad between these two memory protocols, there is only preference according to scenarios. For private memory declaration, as shown in figure 8.2, the whole memory blocks allocated by CPU is copied to another position, which is accessible to GPU only. After GPU executing several kernel functions, the memory between CPU and GPU may differ from each other. Then, if CPU need to access the same memory block, an explicate synchronization is needed before any action, otherwise inconsistent may happen. While for

shared memory, which is shown in figure 8.3, firstly, CPU calls IO Control function for requirements of unified memory, after which a memory handler is returned. Before GPU executing OpenCL kernel, instead of memory copy, memory handler is passed to GPU. When OpenCL kernel finished, no synchronization is needed, as CPU will access same memory slot with GPU's. As system calls are usually cheaper than large memory copy, no matter for execution time or energy consumption, the shared memory protocol is preferred to the private one. However, now it is programmers responsibility to make sure no write-write conflict or read-write conflict happen.

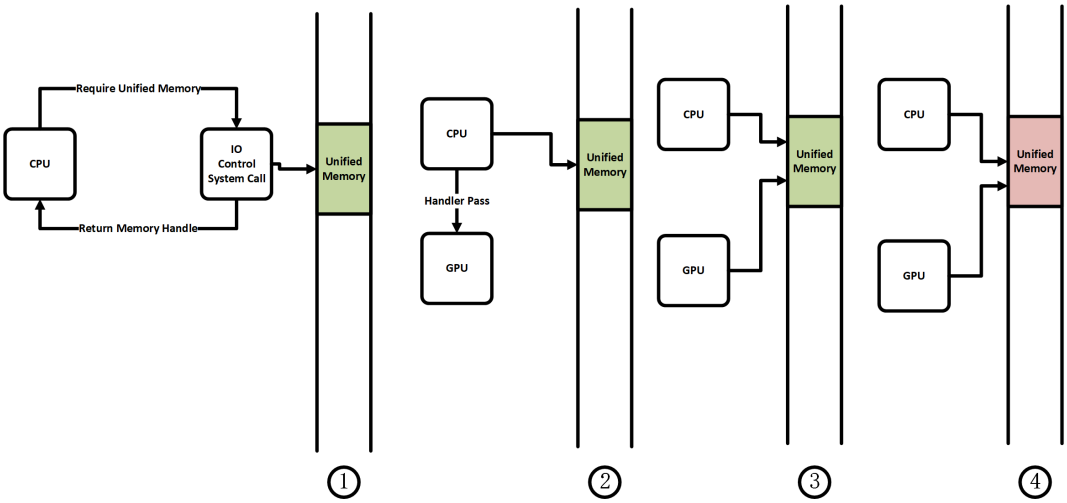


Figure 8.3: Shared Memory on Integrated Memory Architecture

8.3 Implementation Details

There are mainly two methods to declare unified memory [1]. The first one is creating buffer object by `clCreateBuffer` and using `map_over_copy` to make this memory block visible to CPU. The other one is that, for memory shared by GPU and CPU, it is allocated using ION/Gralloc. Then `cl_qcom_host_ptr` extension can be used to create a buffer object, which

maps this ION memory to GPU visible memory by handler passing instead of memory copy. In the official version of Caffe-OpenCL, unified memory is supported by the first method, which is straight and easy-programming.

However, for mobile platforms, it has a serious problem which may even hurt performance. The reason is that, for Caffe, all memory allocation is through a wrapper functioned called `CaffeHostMalloc`, no matter for large memory blocks, such as input images, or small memory blocks such as kernel shape information [8]. It is expensive to allocate small memory by `clCreateBuffer`, as 4K alignment is compulsory, that is, no matter the starting address or the size must be a multiple of 4096. For example, for a memory block storing kernel shape information (three integers: channels, height, width), instead of 12 bytes, 4096 bytes are needed. For desktop software, it is negligible, as 64 GB main memory is quite common, especially for those used for deep learning. While for mobile application, it is intolerable, as even for state-of-the-art mobile phones, 6 GB is rare [55].

According to the above discussion, for mobile-Caffe, the second method is preferred, even though it is harder to manage and more error-prone.

8.4 Experiment Results

We have run experiments for a variety of famous layers, which consist of a wide range of deep learning tasks. The reason why experiments are at the layers level but not neural networks level is that we want to analyze the performance improvement under different computation per memory access. Therefore, layers are divided into three different classes according to

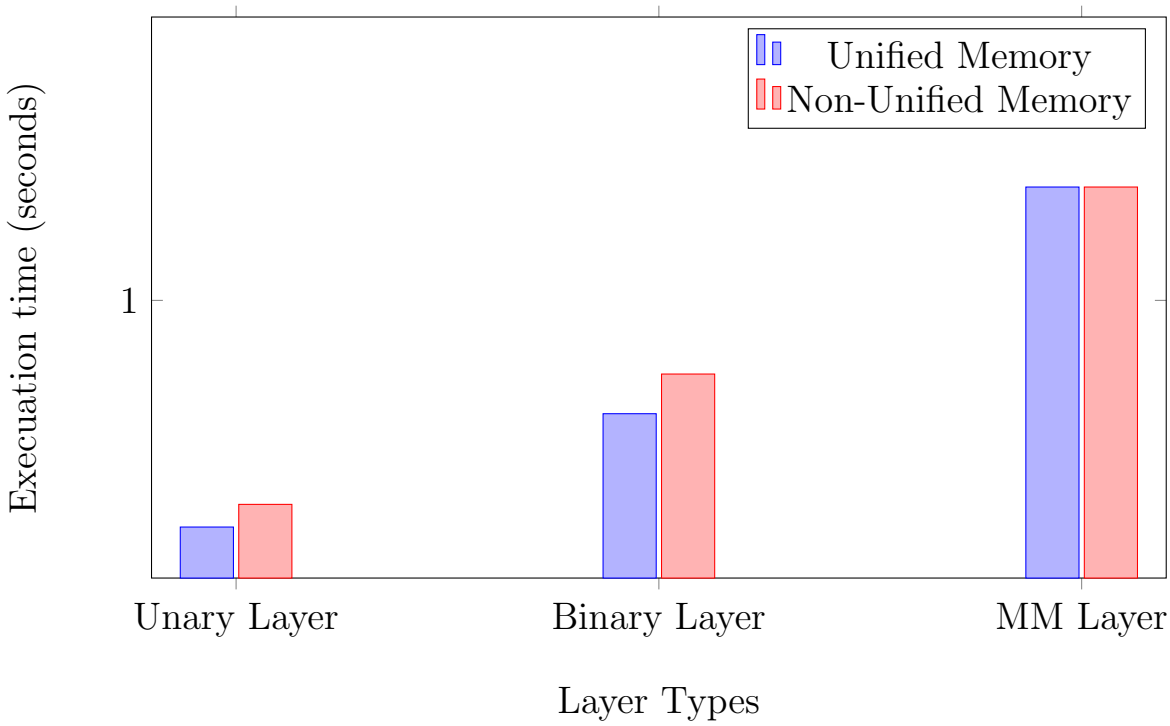


Figure 8.4: Experiments of Inference Time for Different Layers

its $\frac{Computation}{MemoryAccess}$: Unary layers ($Out = Op(In)$) with $\frac{Computation}{MemoryAccess} = 1$, such as rectified linear unit (ReLU) layers; Binary layers ($Out = Op(In_1.In_2)$) with $\frac{Computation}{MemoryAccess} = 2$, such as Eli layers; and Matrix Multiplication layers with $\frac{Computation}{MemoryAccess} = 0$, such as fully connected layers and convolution layers.

All experiments are performed on Xiaomi 6 with Snapdragon 835 and Android 7. During experiments, the battery level and temperature are fixed. All measurements are with random input, and the size is 64 MB, while for Matrix Multiplication layers, the input size reduced to 4 MB to make the running time reasonable. The experiment result is shown in figure 8.4. As expect, Binary layers have the most significant speedup, which is about 9%, as they have the largest $\frac{Computation}{MemoryAccess}$. While for Matrix Multiplication layers, the difference is ignorable as their $\frac{Computation}{MemoryAccess} = 0$.

Chapter 9

Experiments of Real-World Applications and Conclusion

Addition with the benchmarks adopted by the mainstream of deep learning framework benchmarking, there should be real-world applications used to test the performance of our optimization methods. There is only few suitable applications available on Google Store, and most of them do not provide licence for us to modify their code, even for research usage. Therefore, we developed our own mobile deep learning applications, only for demonstration and benchmarking usage, which are art-style transfer and Tang-poems generator. In the chapter, first, discussing these two applications; then presenting experiments results for different optimization; and finally, the whole project is summarized.

9.1 Applications Demonstration

As mentioned in § I, the objective of this project is to port and optimize the traditional deep learning frameworks to offer a fast solution with the help of mobile GPUs. Therefore, in addition to quantitative analysis, we also develop several exciting and fascinating demonstration applications based

on our framework. However, as this project is a research-oriented project, designing mobile apps is out of the scope of this project. Therefore, the UI of these demonstration applications may look very poor.

9.1.1 Art Style Transformation

Recent years, entertainment usages of deep learning technologies have become more and more popular. Among these entertainment applications, style transfer is possibly the most difficult one. Style transfer is about transferring photos to another art styles by separating and recombining the content of the original photos and the art style of some paintings [56]. Different with image processing technologies, which applies simple filters or transformation matrices to change the color or the shape of original images, neural network can extract the art style of the drawing and understand the contents of the photo. After that, the neural network will redraw the photo, which means that the transferred photos may look extremely different with the original ones.

We have ported this fascinating application from desktops to mobile devices, which is shown in figure 9.1. As the neural network is so deep, only convolution layers is listed in the table 9.1. For such a large neural network, even for desktop CPUs (Intel i7 4790), the inference time is about 4.7 seconds. Without neural network compression, the inference time on XiaoMi 6, which is equipped with Snapdragon 835 and 6 GB LPDDR4 main memory, is about 8 seconds.

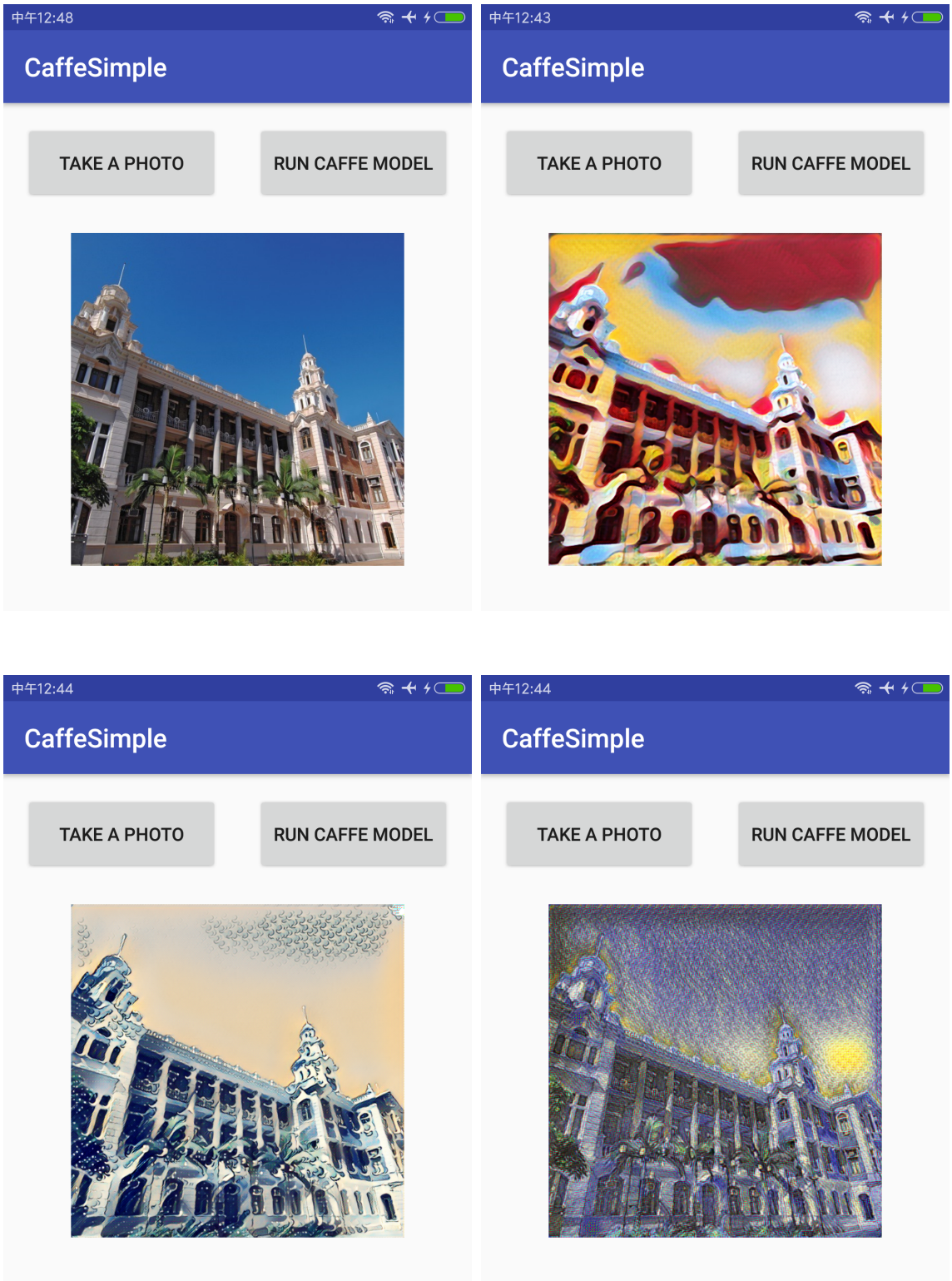


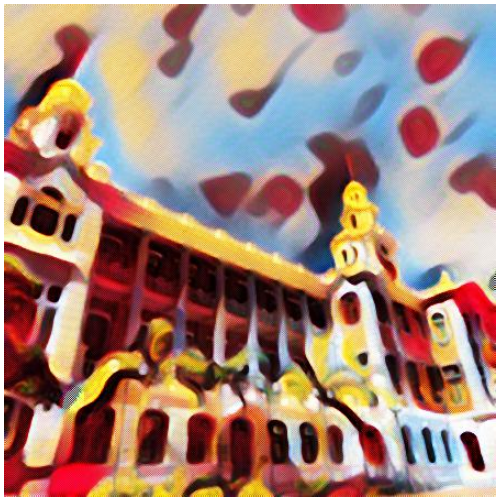
Figure 9.1: Art Style Transfer on Xiaomi 6

Number	Type	C	K	$W_1 \times$H_1$	$W_2 \times$H_2$	$R \times S$
1	CONVOLUTION	3	32	512×512	256×256	9×9
2	CONVOLUTION	32	64	256×256	128×128	3×3
3	CONVOLUTION	64	128	128×128	64×64	3×3
4	RESIDUAL BLOCK	128	128	64×64	64×64	3×3
5	RESIDUAL BLOCK	128	128	64×64	64×64	3×3
6	RESIDUAL BLOCK	128	128	64×64	64×64	3×3
7	RESIDUAL BLOCK	128	128	64×64	64×64	3×3
8	RESIDUAL BLOCK	128	128	64×64	64×64	3×3
9	CONVOLUTION	128	64	64×64	128×128	3×3
10	CONVOLUTION	64	32	128×128	256×256	3×3
11	CONVOLUTION	32	3	256×256	512×512	9×9

Table 9.1: Architecture of Fast Style Transfer

As the neural network is relative large, we choose K-Means Clustering compression to speed up the application. The compression ratio is chosen as 50% and 62.5%, as it gives reasonable accuracy loss and speed up for most deep neural networks. The result of the transferred images and the inference time is shown is figure 9.2 and figure 9.3.

(a) Compression Rate 0; Inference Time: 7.8s



(b) Compression Rate 50%; Inference Time: 3.7s

(c) Compression Rate 67.5%; Inference Time: 3.4s

Figure 9.2: Candy Style Transfer with Different Compression Rate

9.1.2 Tang Poems Generator

Chinese classical poetry is undoubtedly one of the most beautiful cultural heritage even all over the world, which significantly influences the Chinese people and culture, especially for the ancient time. There are several researches about generating Chinese poems automatically, which start from 1960s. However, most of these early attempts use statistic methods, by which

(a) Compression Rate 0; Inference Time: 7.8s

(b) Compression Rate 50%; Inference Time: 3.6s

(c) Compression Rate 67.5%; Inference Time: 3.4s

Figure 9.3: Star Night Style Transfer with Different Compression Rate

it is difficult to generate meaningful sentences.

With the development of deep learning technology, natural language processing has also gain impressive improvements. A recent study applied a famous deep neural network, named recurrent neural network, to Chinese poems generator [57]. As a typical natural language processing application,

Chinese poems generator is also ported to mobile platforms by us. This generator generates Chinese poem automatically by a initial character, except the deep learning model and thew initial character, there is no other information provided to the application. As the core operations of recurrent neural network are matrix multiplications, which can be significantly speedup by our optimization.

The famous Chinese poems data-set: Chinese-poetry [58] is used to fine-tuning the neural network during compression under MIT license. Several sample poems generated by our application are shown in figure 9.4.

9.1.3 Speedup for Different Applications with Different Optimization

All experiments are performed on the most powerful mobile processor: Snapdragon 835 with Adreno 540 GPU. Instead of using development board, real-world mobile phone (XiaoMi 6) is used both for better illustration and demonstration. The experiment results are shown in figure 9.5. It can be seen that for images related tasks, Unified Memory and Half Precision Matrix Multiplication do not help a lot. However, deep neural network compression gives significant speed up (more than 2×). On the other hand, for natural language processing, whose core operation is matrix multiplication, only the half precision gives speedup, while all other optimization methods slow down the inference stage.

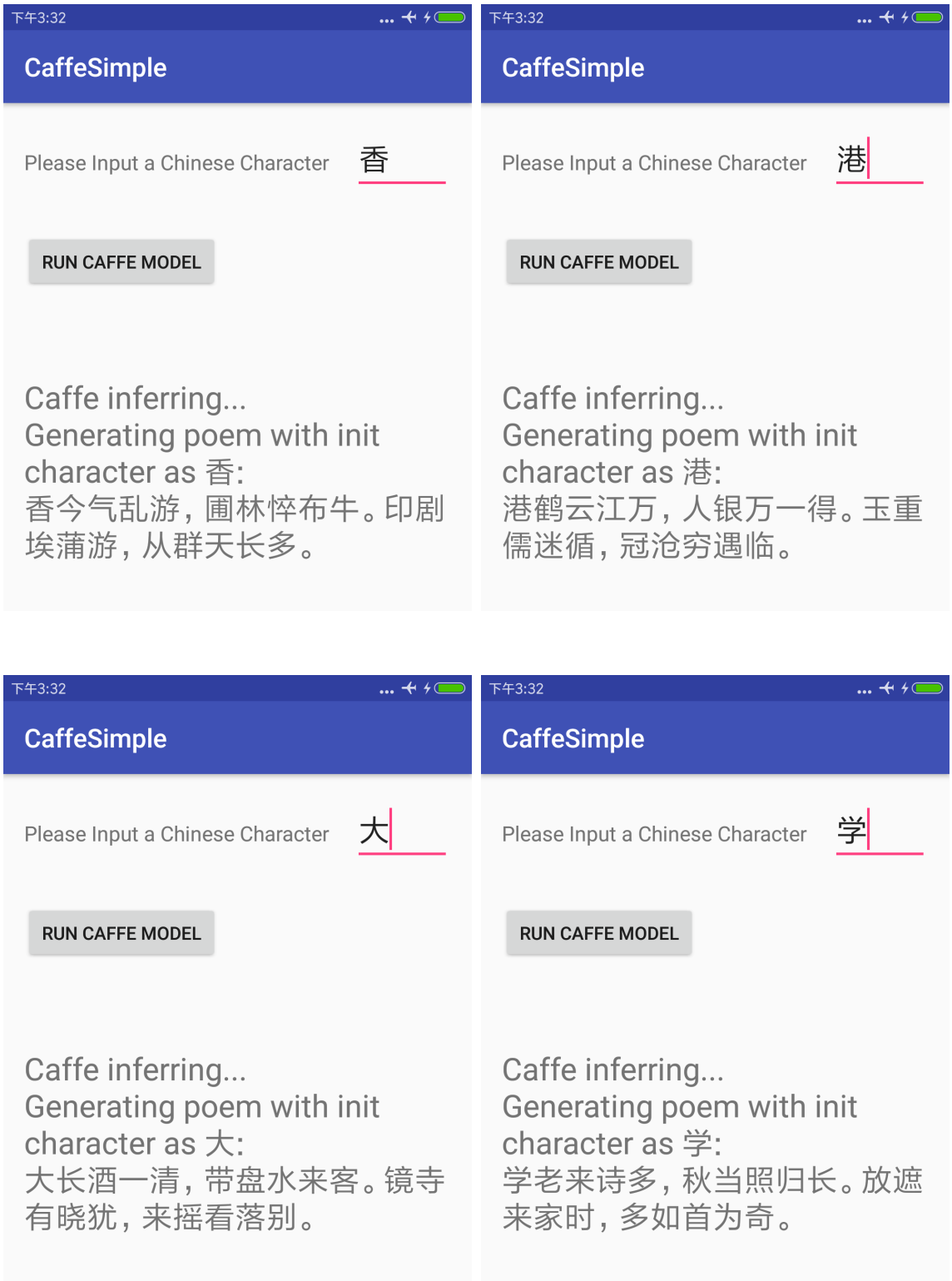


Figure 9.4: Tang Poems Generator on Xiaomi 6

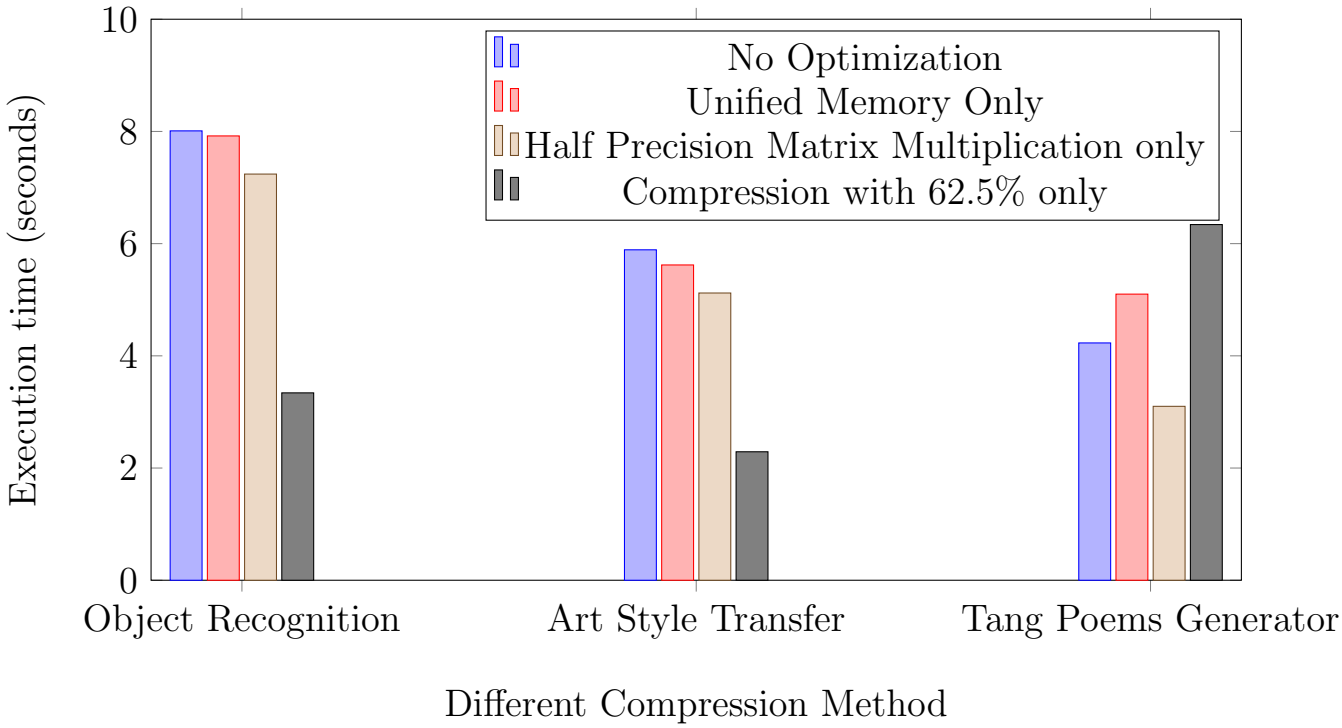


Figure 9.5: Experiments of Inference Time for Different Layers

9.2 Conclusion

In this project, we have proposed a toolkit named MobileDL for mobile platforms. MobileDL integrates Caffe and LIBDNN to overcome the problem caused by limited memory storage and bandwidth. Unlike previous approaches that focused on storage reduction, MobileDL explored methods to reduce execution time and power consumption. Together with three novel algorithms introduced in § V, VII, VIII, empirical results suggested that MobileDL achieved up to 3× speedup relative to the state-of-the-art mobile deep learning framework with only minor loss of accuracy.

Bibliography

[1] I. CQualcomm Technologies, "Qualcomm® snapdragon™ mobile platform opencl general programming and optimization," 2017.

[2] A. Canziani, A. Paszke, and E. Culurciello, "An analysis of deep neural network models for practical applications," *arXiv preprint arXiv:1605.07678*, 2016.

[3] P. M. B. Ginsburg, S. Nikolaev, "Training with mixed precision," *GPU Technology Conference, 2017*, 2017.

[4] Mar 2017. [Online]. Available: https://developer.apple.com/library/content/documentation/3DDrawing/Concept

[5] Y. LeCun *et al.*, "Lenet-5, convolutional neural networks," *URL: http://yann. lecun. com/exdb/lenet*, 2015.

[6] P. Micikevicius, S. Narang, J. Alben, G. Diamos, E. Elsen, D. Garcia, B. Ginsburg, M. Houston, O. Kuchaev, G. Venkatesh *et al.*, "Mixed precision training," *arXiv preprint arXiv:1710.03740*, 2017.

[7] K.-T. Cheng and Y.-C. Wang, "Using mobile gpu for general-purpose computing–a case study of face recognition on smartphones," in *VLSI Design, Automation and Test (VLSI-DAT), 2011 International Symposium on*. IEEE, 2011, pp. 1–4.

[8] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell, "Caffe: Convolutional architecture for fast feature embedding," in *Proceedings of the 22nd ACM international conference on Multimedia.* ACM, 2014, pp. 675–678.

[9] G. Hinton, L. Deng, D. Yu, G. E. Dahl, A.-r. Mohamed, N. Jaitly, A. Senior, V. Vanhoucke, P. Nguyen, T. N. Sainath *et al.*, "Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups," *IEEE Signal Processing Magazine*, vol. 29, no. 6, pp. 82–97, 2012.

[10] R. Collobert and J. Weston, "A unified architecture for natural language processing: Deep neural networks with multitask learning," in *Proceedings of the 25th international conference on Machine learning.* ACM, 2008, pp. 160–167.

[11] N. D. Lane, S. Bhattacharya, P. Georgiev, C. Forlivesi, L. Jiao, L. Qendro, and F. Kawsar, "Deepx: A software accelerator for low-power deep learning inference on mobile devices," in *Information Processing in Sensor Networks (IPSN), 2016 15th ACM/IEEE International Conference on.* IEEE, 2016, pp. 1–12.

[12] Y. Gong, L. Liu, M. Yang, and L. Bourdev, "Compressing deep convolutional networks using vector quantization," *arXiv preprint arXiv:1412.6115*, 2014.

[13] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krüger, A. E. Lefohn, and T. J. Purcell, "A survey of general-purpose computation on

graphics hardware," in *Computer graphics forum*, vol. 26, no. 1. Wiley Online Library, 2007, pp. 80–113.

[14] A. Pathania, Q. Jiao, A. Prakash, and T. Mitra, "Integrated cpu-gpu power management for 3d mobile games," in *Proceedings of the 51st Annual Design Automation Conference*. ACM, 2014, pp. 1–6.

[15] "Snapdragon 835 mobile platform with 10 nm 64-bit cpu," Aug 2017. [Online]. Available: https://www.qualcomm.com/products/snapdragon/processors/835

[16] Arm, "Mali gpu – arm." [Online]. Available: https://www.arm.com/products/graphics-and-multimedia/mali-gpu

[17] "Nvidia tegra: The world's fastest mobile processors." [Online]. Available: http://www.nvidia.com/object/tegra.html

[18] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 770–778.

[19] A. Parashar, M. Rhu, A. Mukkara, A. Puglielli, R. Venkatesan, B. Khailany, J. Emer, S. W. Keckler, and W. J. Dally, "Scnn: An accelerator for compressed-sparse convolutional neural networks," 2017.

[20] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in neural information processing systems*, 2012, pp. 1097–1105.

[21] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *arXiv preprint arXiv:1409.1556*, 2014.

[22] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going deeper with convolutions," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2015, pp. 1–9.

[23] M. Denil, B. Shakibi, L. Dinh, N. de Freitas *et al.*, "Predicting parameters in deep learning," in *Advances in Neural Information Processing Systems*, 2013, pp. 2148–2156.

[24] R. Gray, "Vector quantization," *IEEE Assp Magazine*, vol. 1, no. 2, pp. 4–29, 1984.

[25] S. Arnborg and A. Proskurowski, "Linear time algorithms for np-hard problems restricted to partial k-trees," *Discrete applied mathematics*, vol. 23, no. 1, pp. 11–24, 1989.

[26] T. Kanungo, D. M. Mount, N. S. Netanyahu, C. D. Piatko, R. Silverman, and A. Y. Wu, "An efficient k-means clustering algorithm: Analysis and implementation," *IEEE transactions on pattern analysis and machine intelligence*, vol. 24, no. 7, pp. 881–892, 2002.

[27] K. Alsabti, S. Ranka, and V. Singh, "An efficient k-means clustering algorithm," 1997.

[28] L. Bottou, "Large-scale machine learning with stochastic gradient descent," in *Proceedings of COMPSTAT'2010*. Springer, 2010, pp. 177–186.

[29] F. Cheng, "Meet the snapdragon 835: a next-gen processor made for power users," *Qualcomm Snapdragon Blog*, 2017.

[30] "Mi 6 picture perfect dual camera." [Online]. Available: http://www.mi.com/en/mi6/

[31] "iphone x." [Online]. Available: https://www.apple.com/iphone-x/

[32] J. Standard, "Lpddr sdram standard," *Revision of JESD209-2E*, 2010.

[33] "Huawei p10 smartphone — mobile phones — huawei global," Mar 2017. [Online]. Available: http://consumer.huawei.com/en/phones/p10/

[34] "It's here: The new geforce gtx 1080ti graphics card." [Online]. Available: https://www.nvidia.com/en-us/geforce/products/10series/geforce-gtx-1080-ti/

[35] "Radeon™ rx vega." [Online]. Available: https://gaming.radeon.com/en/product/vega/radeon-rx-vega-64/

[36] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin *et al.*, "Tensorflow: Large-scale machine learning on heterogeneous distributed systems," *arXiv preprint arXiv:1603.04467*, 2016.

[37] R. Collobert, K. Kavukcuoglu, and C. Farabet, "Torch7: A matlab-like environment for machine learning," in *BigLearn, NIPS Workshop*, no. EPFL-CONF-192376, 2011.

[38] "Delivering real-time ai in the palm of your hand." [Online]. Available: https://code.facebook.com/posts/196146247499076/delivering-real-time-ai-in-the-palm-of-your-hand/

[39] D. Team, "Deeplearning4j: Open-source distributed deep learning for the jvm," *Apache Software Foundation License*, vol. 2, 2016.

[40] sh1r0, "sh1r0/caffe-android-demo," Dec 2016. [Online]. Available: https://github.com/sh1r0/caffe-android-demo

[41] A. Mathur, N. D. Lane, S. Bhattacharya, A. Boran, C. Forlivesi, and F. Kawsar, "Deepeye: Resource efficient local execution of multiple deep vision models using wearable commodity hardware," 2017.

[42] S. S. Latifi Oskouei, H. Golestani, M. Hashemi, and S. Ghiasi, "Cnndroid: Gpu-accelerated execution of trained deep convolutional neural networks on android," in *Proceedings of the 2016 ACM on Multimedia Conference*. ACM, 2016, pp. 1201–1205.

[43] L. N. Huynh, Y. Lee, and R. K. Balan, "Deepmon: Mobile gpu-based deep learning framework for continuous vision applications," in *Proceedings of the 15th Annual International Conference on Mobile Systems, Applications, and Services*. ACM, 2017, pp. 82–95.

[44] E. L. Denton, W. Zaremba, J. Bruna, Y. LeCun, and R. Fergus, "Exploiting linear structure within convolutional networks for efficient evaluation," in *Advances in Neural Information Processing Systems*, 2014, pp. 1269–1277.

[45] M. Jaderberg, A. Vedaldi, and A. Zisserman, "Speeding up convolutional neural networks with low rank expansions," *arXiv preprint arXiv:1405.3866*, 2014.

[46] V. Lebedev, Y. Ganin, M. Rakhuba, I. Oseledets, and V. Lempitsky, "Speeding-up convolutional neural networks using fine-tuned cp-decomposition," *arXiv preprint arXiv:1412.6553*, 2014.

[47] Y.-D. Kim, E. Park, S. Yoo, T. Choi, L. Yang, and D. Shin, "Compression of deep convolutional neural networks for fast and low power mobile applications," *arXiv preprint arXiv:1511.06530*, 2015.

[48] J. Wu, C. Leng, Y. Wang, Q. Hu, and J. Cheng, "Quantized convolutional neural networks for mobile devices," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2016, pp. 4820–4828.

[49] T. Chen, M. Li, Y. Li, M. Lin, N. Wang, M. Wang, T. Xiao, B. Xu, C. Zhang, and Z. Zhang, "Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems," *arXiv preprint arXiv:1512.01274*, 2015.

[50] K. Rupp, P. Tillet, F. Rudolf, J. Weinbub, A. Morhammer, T. Grasser, A. Jungel, and S. Selberherr, "Viennacl—linear algebra library for multi- and many-core architectures," *SIAM Journal on Scientific Computing*, vol. 38, no. 5, pp. S412–S439, 2016.

[51] Y. Sun, Y. Chen, X. Wang, and X. Tang, "Deep learning face representation by joint identification-verification," in *Advances in neural information processing systems*, 2014, pp. 1988–1996.

[52] S. Han, J. Pool, J. Tran, and W. Dally, "Learning both weights and connections for efficient neural network," in *Advances in neural information processing systems*, 2015, pp. 1135–1143.

[53] "Introducing nvidia titan v: The world's most powerful pc graphics card." [Online]. Available: https://www.nvidia.com/en-us/titan/titan-v/

[54] "Nvidia volta ai architecture." [Online]. Available: https://www.nvidia.com/en-us/data-center/volta-gpu-architecture/

[55] "galaxy-s8." [Online]. Available: https://www.samsung.com/us/explore/galaxy-s8/

[56] J. Johnson, A. Alahi, and L. Fei-Fei, "Perceptual losses for real-time style transfer and super-resolution," in *European Conference on Computer Vision*. Springer, 2016, pp. 694–711.

[57] X. Yi, R. Li, and M. Sun, "Generating chinese classical poems with rnn encoder-decoder," in *Chinese Computational Linguistics and Natural Language Processing Based on Naturally Annotated Big Data*. Springer, 2017, pp. 211–223.

[58] gao, "Poetry of grammar and grammar of poetry," *Poetry of Grammar and Grammar of Poetry*, 2017. [Online]. Available: https://github.com/chinese-poetry/chinese-poetry