Deep Learning on Mobile GPUs
A Fast and Energy Efficient Solution

JI Zhuoran 3035139915

The University of Hong Kong

An Interim Report Submitted for Review

20 Jan 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 Haicheng WANG, for his kind support regarding the test case and data. I am grateful for help received from Dr. Tam for his valuable comments. Last but not least, I want to express my appreciation towards Zhihan CHEN, who lent equipment to me at the very early stage.

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 power efficiency, 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) Zero-Copy; (2) Convolutional Neural Network Compression and (3), Half Precision Computation Supporting. Experiments on several famous benchmarks demonstrated about $3\times$ *speed-up with merely* 1% *loss of classification accuracy. Additionally, MobileDL also achieves significant energy usage reduction, which is critical to battery-power 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 optimization methods, namely: (1) Zero-Copy; (2) Convolution Neural Network Compression and (3) Half Precision Support. Through these three optimization, MobileDL offers a fast and energy-efficiency 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; then presenting three novel methodologies (§ IV, V, VI), and finally future work will be discussed (§ VII).

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.

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.

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

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 GPUs, the gap of memory latency and bandwidth is extremely significant, as host memory is usually limited and extremely slow, for example, in Table 2.3, 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.

Table 2.2: My caption

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 2.3: Comparison on various of GPUs

2.2.2 GPU Scheduling

For Adreno GPUs of Android platforms, context switch is enabled, that is a high priority task can preemptive a low priority process if necessary [1]. For example, a UI rendering task can pause a computation task if it runs a long time that makes UI be lagging and unresponsive (see figure 2.1). The context switch is expensive, especially on GPUs. However, it is critical to some mobile computing scenario, such as autopilot cars. To avoid this penalty, it is better to have small workgroups for computation kernels, so that it will be finished before the context switch. Meanwhile, it is apparent that adopting this scheduling strategy regardless working-environment is not the best solution. For example, for the autopilot car, sometimes, we do not care about UI rendering during driving, instead, computation kernels should be the highest priority tasks.

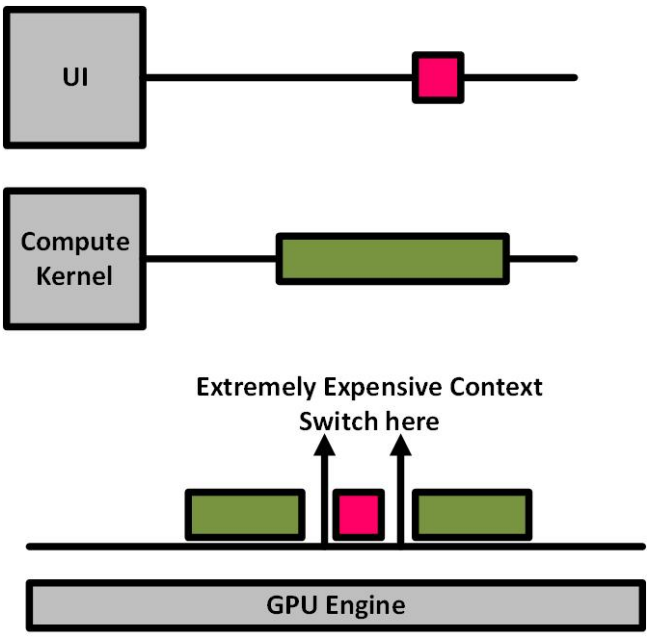


Figure 2.1: UI rendering tasks preemptive computation kernels

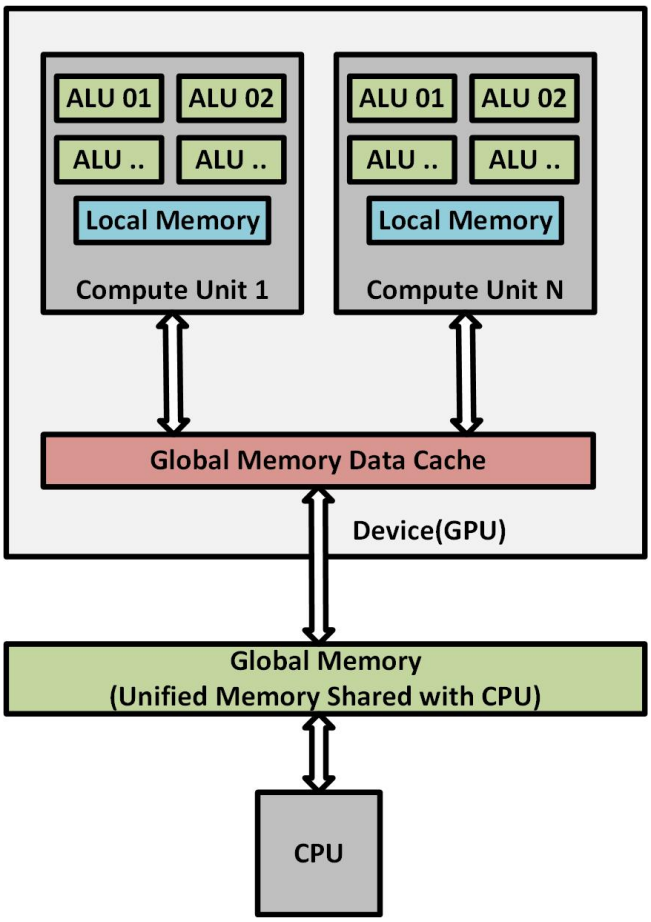


Figure 2.2: Memory architecture of Adreno 5XX family

2.2.3 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.

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.

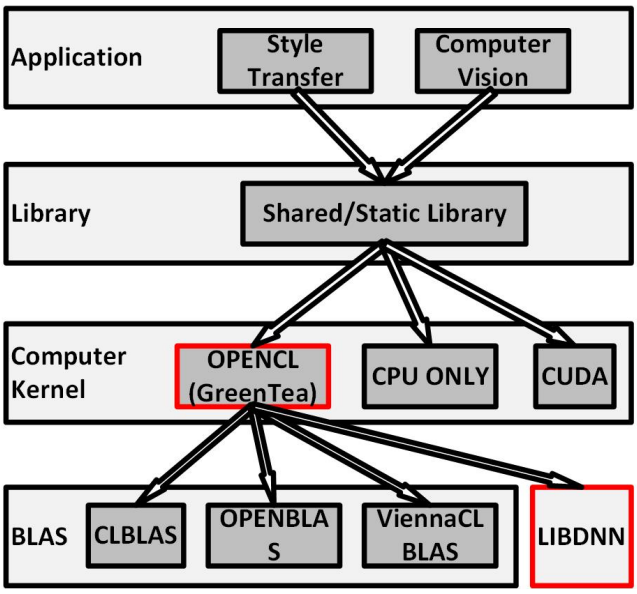


Figure 4.1: Software Architecture of Caffe

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.

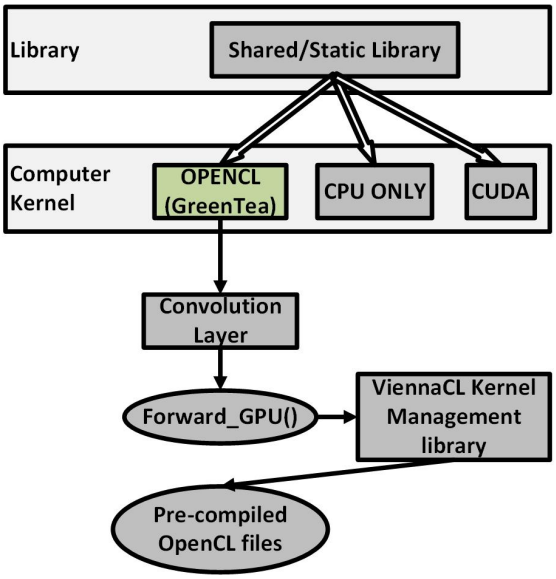


Figure 4.2: Caffe-Greentra-GPU

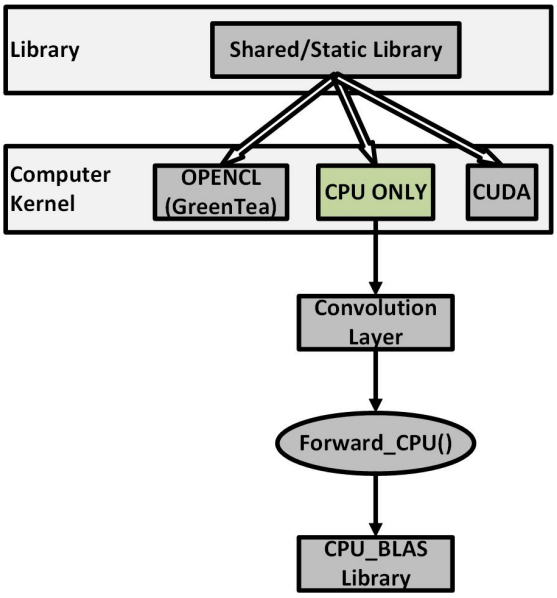


Figure 4.3: Caffe-CPU

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.

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. Although we are not interested in this implantation, the reason for the poor performance is going to be found out in the next semester.

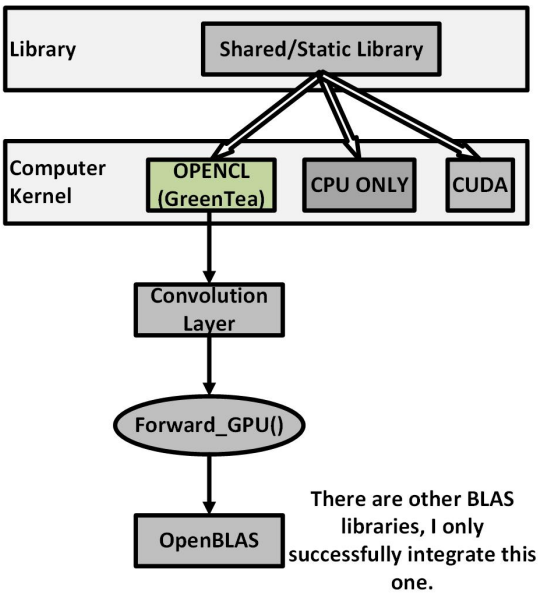


Figure 4.4: Caffe-Greentea-BLAS

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

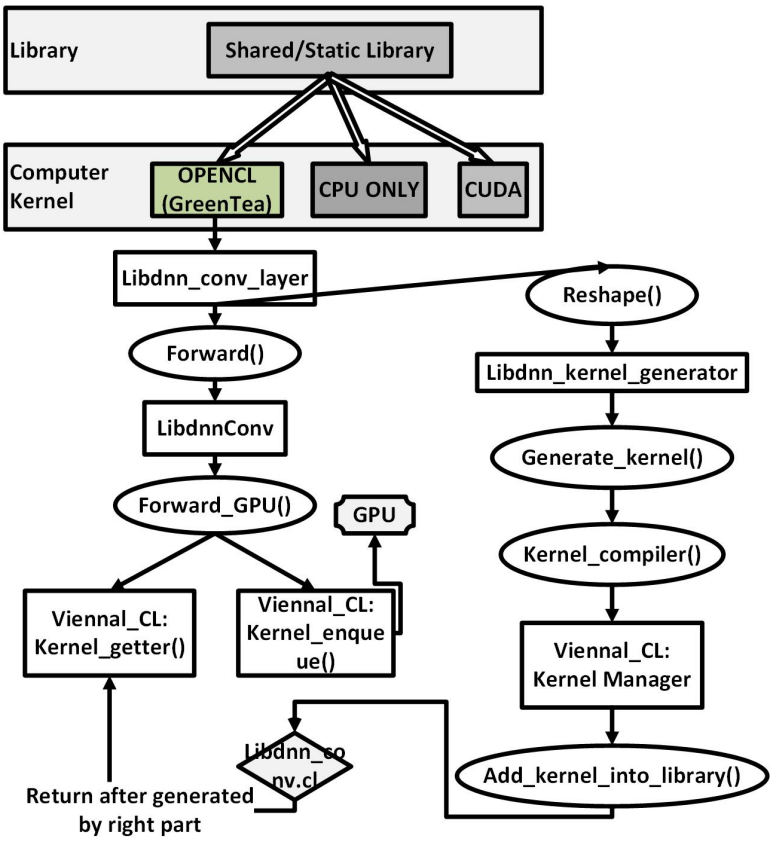


Figure 4.5: Caffe-LIBDNN

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.

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 4.6, which satisfies the expectations in the previous sections.

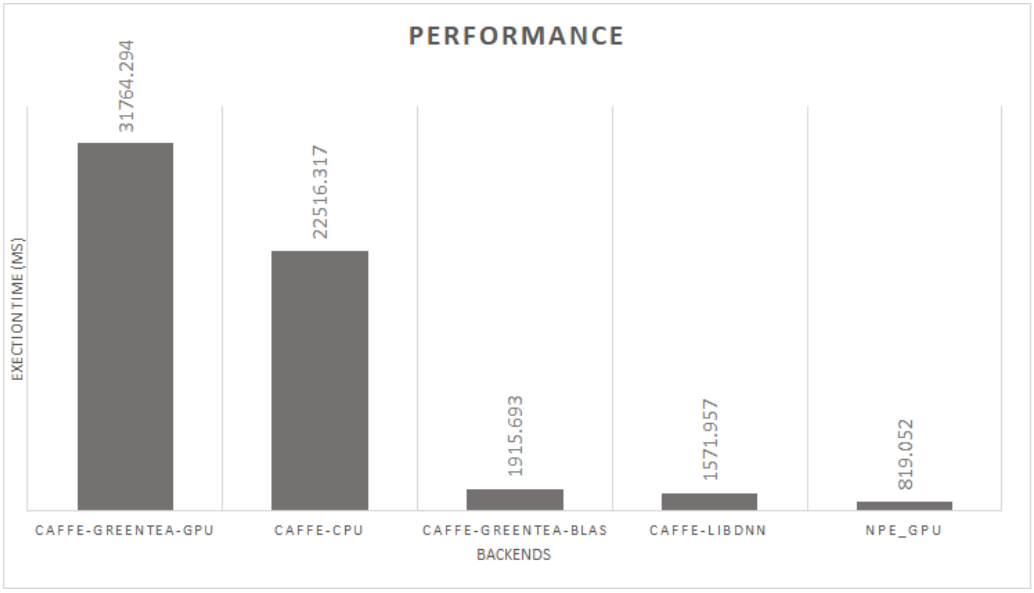


Figure 4.6: Performance of different Backends

Chapter 5

Unified Memory: Zero Copy between GPUs and CPUs

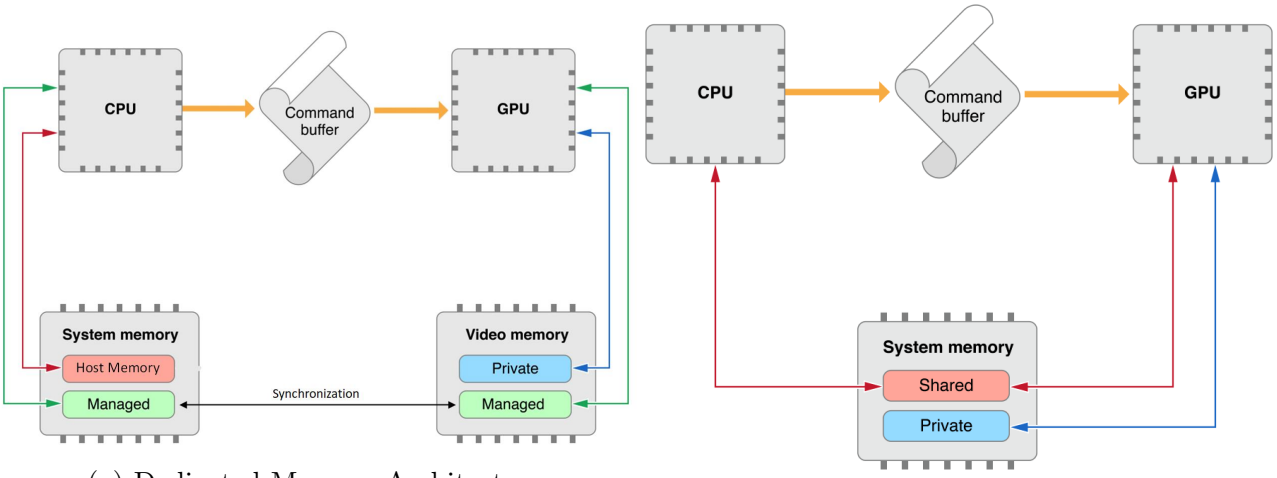
Memory copy is extremely expensive and energy-consuming. According to the standards specification of Low Power DDR (LPDDR) [32], the power consumption is 40 pJ/bit. However, for deep learning applications, for each layer, hundreds MB of data will be copied, even worse, most of the mobile deep learning 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.

5.1 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 5.1.

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 mem-



(a) Dedicated Memory Architecture

(b) Integrated Memory Architecture

Figure 5.1: Memory Architecture of Desktop Platforms and Mobile Platforms [2]

ory 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.

5.2 Memory Management Protocols

As shown in figure 5.1b, 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 5.2, 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 5.3, 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.

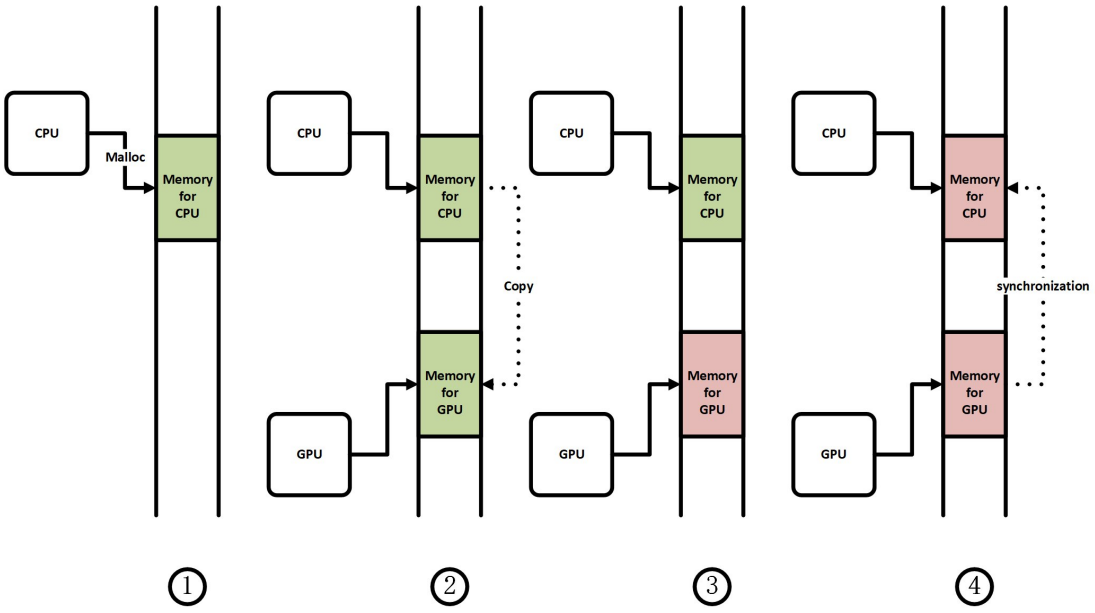


Figure 5.2: Private Memory on Integrated Memory Architecture

5.3 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 [51].

According to the above discussion, for mobile-Caffe, the second method is preferred, even

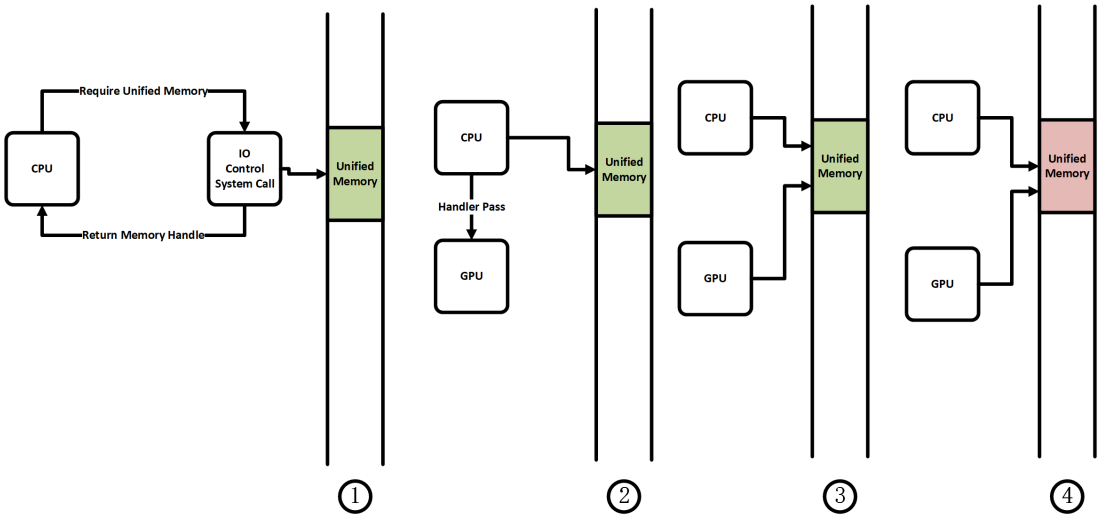


Figure 5.3: Shared Memory on Integrated Memory Architecture

though it is harder to manage and more error-prone.

5.4 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 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 5.4. 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 $\frac{Computation}{MemoryAccess} = 0$.

However, compare to speed up, memory usage reduction and energy-saving are more critical benefits of this technique, as MobileDL executes on mobile platforms. The experiments for

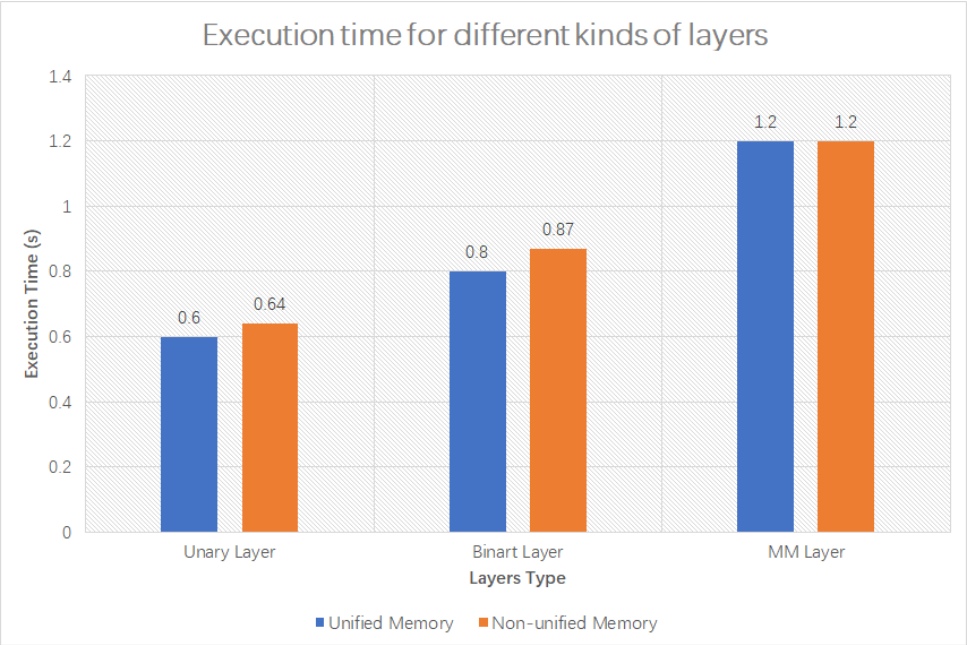


Figure 5.4: Experiments of Inference Time for Different Layers

energy-saving are going to be performed in the second semester.

Chapter 6

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] [52]. 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.

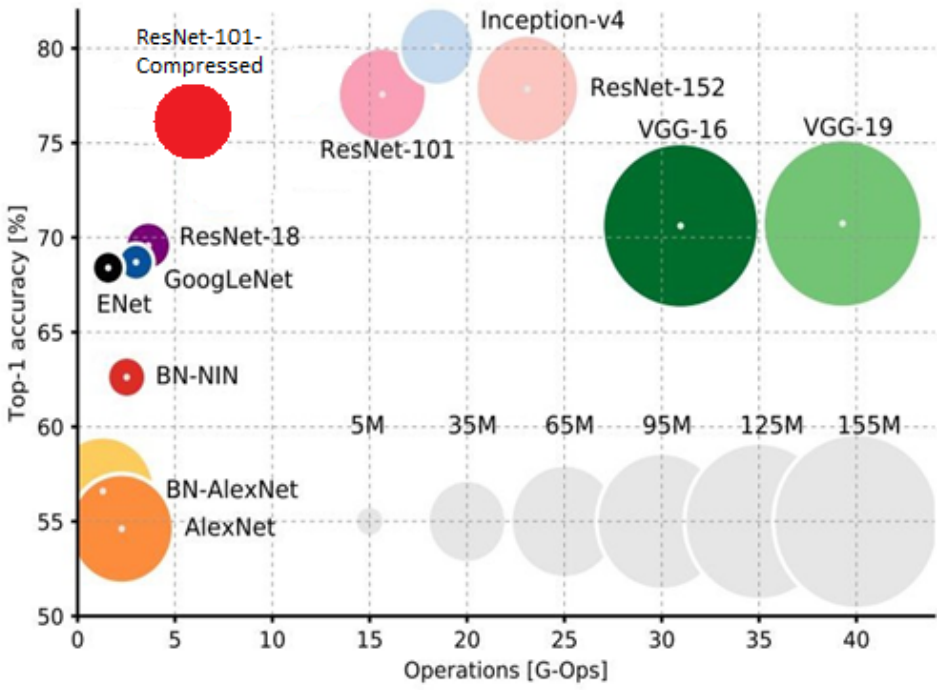


Figure 6.1: Relationship between the classification accuracy and number of operations [3]

On the other hand, the improvement of the performance is not proportional to the growth of the complexity. Canziani et al. [3] listed the relationship between the classification accuracy and number of operations, as shown in figure 6.1. 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 better than ResNet-18 (see figure 6.1). According to the above assumption, we proposed convolution neural networks compression technique for mobile platforms.

6.1 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.

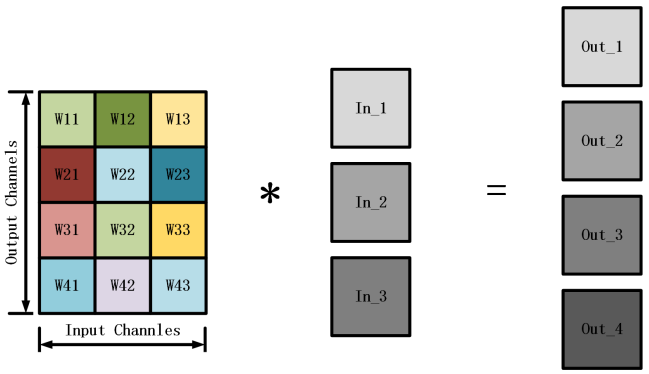


Figure 6.2: A brief illustration of convolution layers

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 6.2. 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 6.3).

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

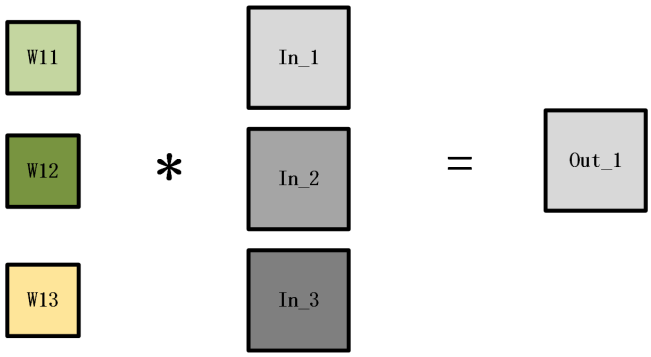


Figure 6.3: Convolution for same output channels

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 6.2, 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.

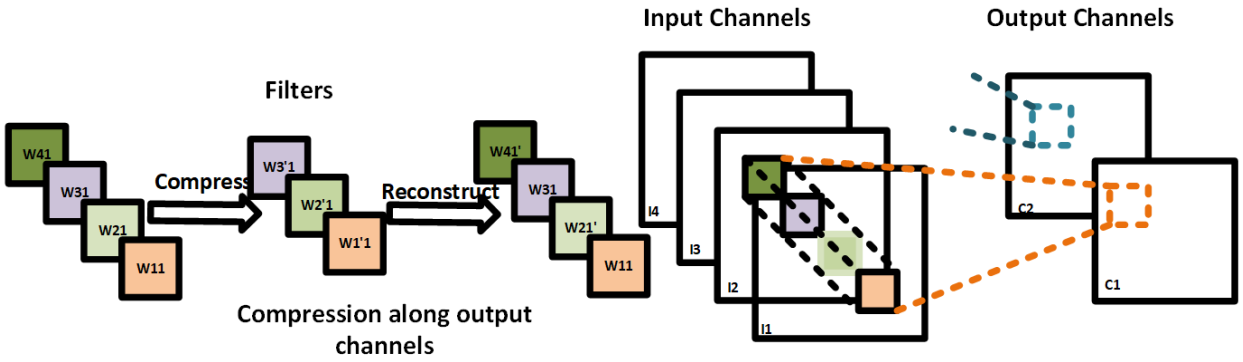


Figure 6.4: Compression along input channels

The reason is shown in figure 6.4, 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.

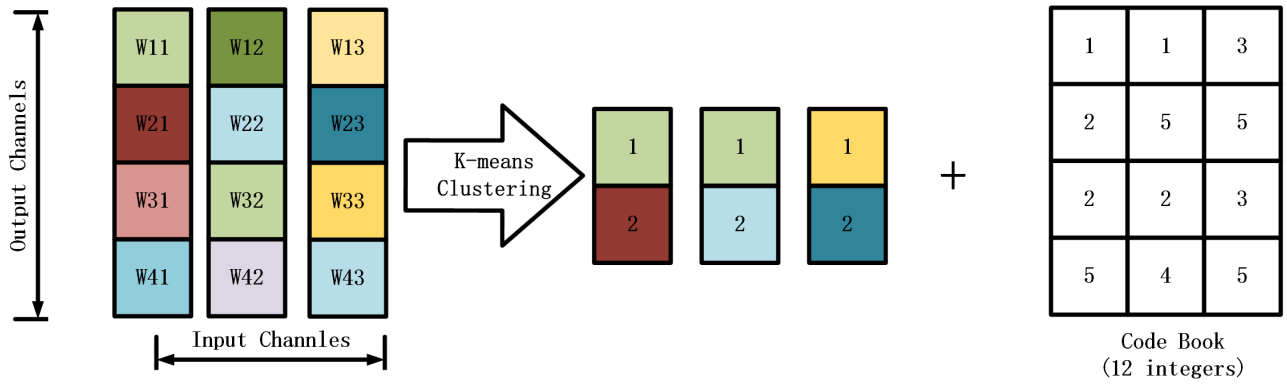


Figure 6.5: Convolution Computation:

6.2 Algorithm

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

6.2.1 Neural Network Minimization

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, NNM 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 6.5. 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:

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

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 mathematically expressed as

$$O = O'M$$

As a result, the overall time complexity will be reduced from $\mathcal{O}(C \times K \times R \times S \times W \times H)$ to $\mathcal{O}(C \times \tilde{K} \times R \times S \times W \times H + K \times W \times H)$. 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 if figure 6.4.

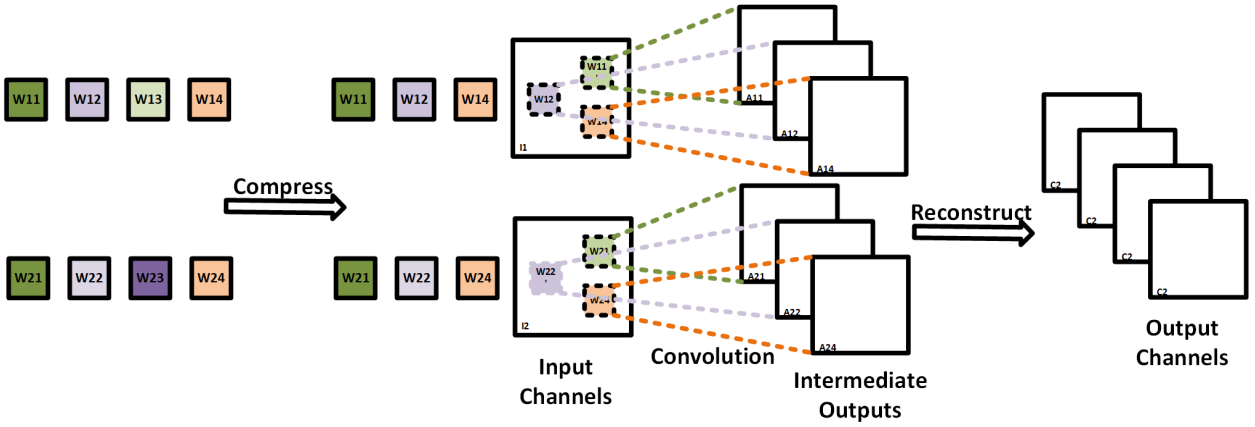


Figure 6.6: Compression along output channels

6.2.2 Converge Points Transition

With NNM, 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 NNM is usually not the one giving the best classification accuracy after NNM. Furthermore, as NNM 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 NNM.

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 regularization. The updating function then can be expressed as $\theta = \theta - \epsilon(g + \alpha\nabla\Omega(\theta))$, where g

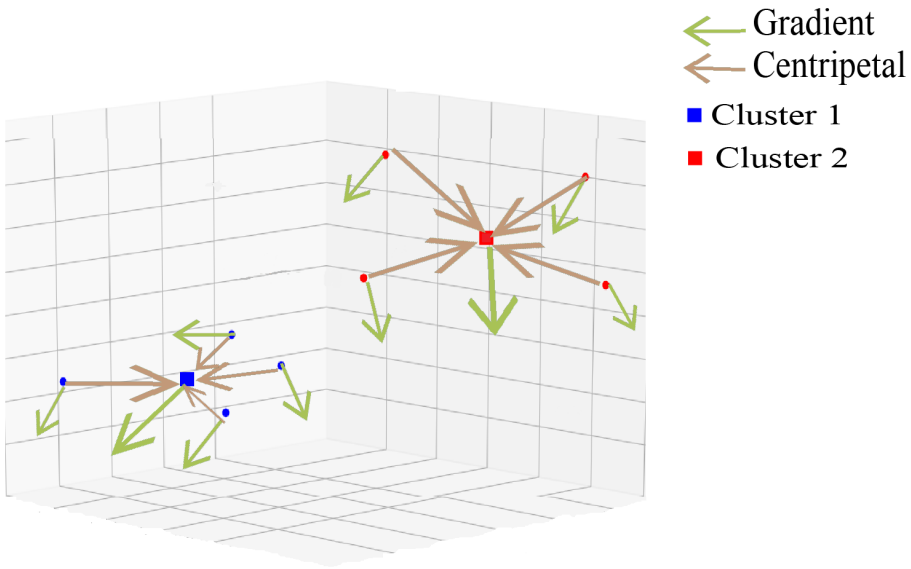


Figure 6.7: Converge Points Transition

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

$$\Psi(\theta) = reshape\left\{\tilde{S}_1 M_1, \tilde{S}_2 M_2, ..., \tilde{S}_c M_c\right\} - \theta$$

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 6.7. 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

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]

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.

6.3 Experiments and Results

6.3.1 Experiment on LeNet

The MNIST database was used to evaluate the performance of MoblieDL on LeNet. The MNIST database contains 60,000 training images and 10,000 testing images of hand-written digits, and the detailed architecture of LeNet is shown in Table 6.1, in which there are two convolution layers. The neural network was pre-trained with different compression rates with and without CPT, then NNM was used to minimize the original neural network. The results are reported in Figure 6.9, Figure 6.8 for speedup and classification accuracy, respectively.

6.3.1.1 Classification error for different compression rate

Figure 6.8 shows the classification error of the LeNet on MNIST dataset with different compression rates. The neural network without compression is chosen as the baseline. If CPT is not adopted, even little compression will lead to extremely high classification error.

In our experiment, a compression rate of 25% will cause about 11× classification error (see Figure 6.8), in other words, statistically, for any misclassification of the original neural network, there will be 11 digits images be classified incorrectly by the compressed neural network. The classification error is too high to accept, even for mobile application, which is less critical than professional software. By incorporating the CPT algorithm, surprisingly, much high compression was achieved with a merely minor loss in accuracy. For LeNet on MNIST dataset, MobileDL achieved up to 50% compression rate with less than 1% loss in classification accuracy.

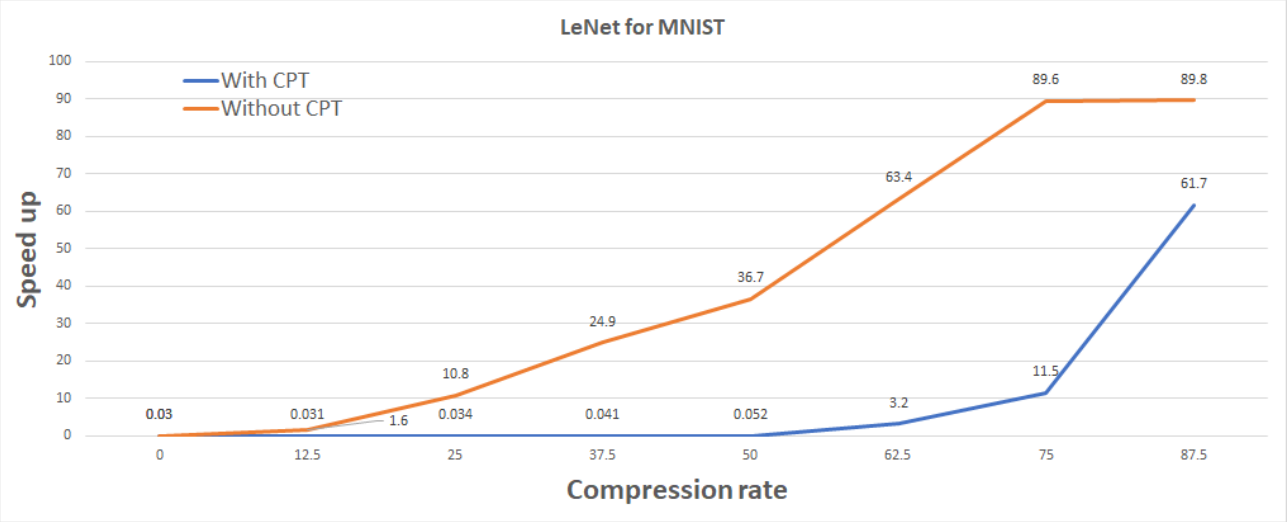


Figure 6.8: Classification Error of LeNet

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. However, this is just a conjecture. In the rest of this project, more reasons are going to be explored, and formal proof will be given.

6.3.1.2 Speed up for different compression rate

Figure 6.9 shows the empirical speedup we achieved on the Snapdragon 835 platform. Theoretically, if the compression rate is 50%, as there is only half of computation needed, at most 2× speedup should be achieved.

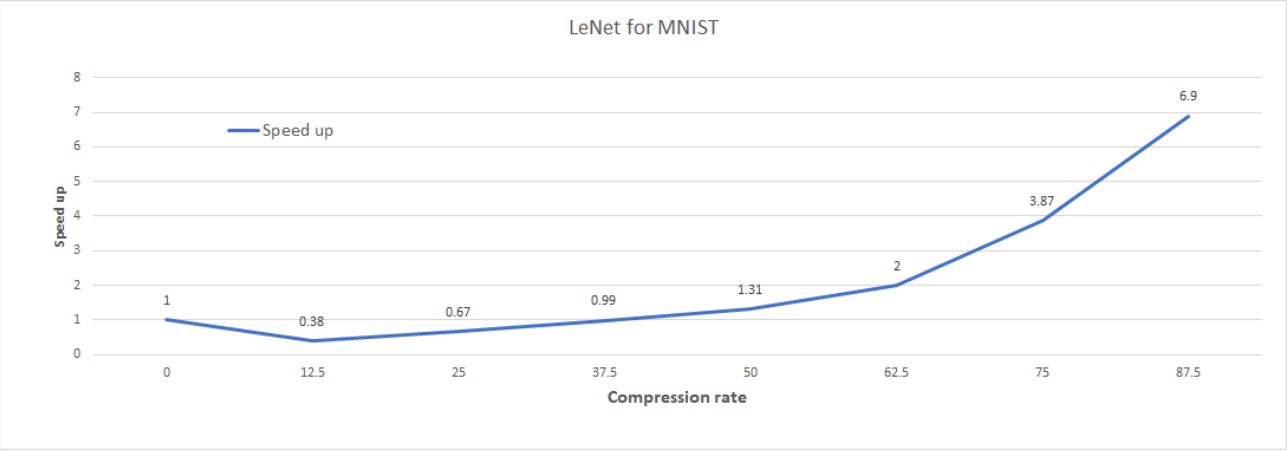


Figure 6.9: Speed up of LeNet

However, it can be seen from Figure 6.9 that the speedup is 1.31×, which is much less than 2×. The reason for 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 6.6, 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 1.45× speedup, as the memory access is much more expensive than computation on integrated memory architecture.

In the second semester, we are going to run more experiments on other famous deep neural networks, while the more detailed profile of memory behaviour will be given.

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.

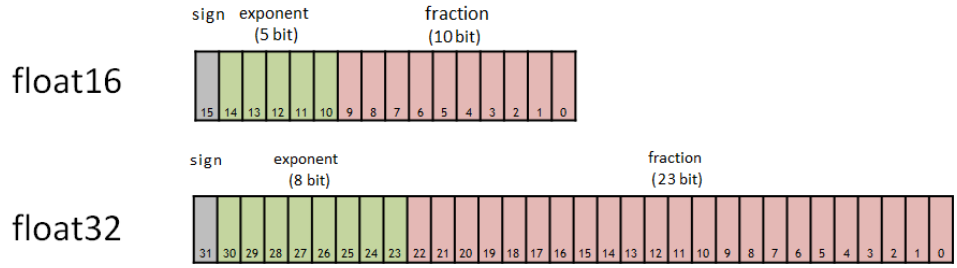


Figure 7.1: Bits usage of Float16 and Float32 in IEEE754

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	Matrix Multiplication 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.

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

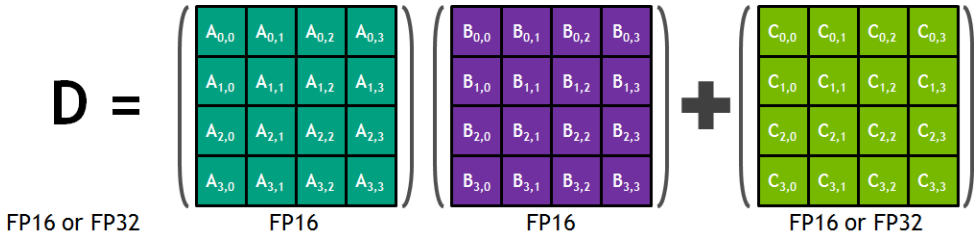


Figure 7.2: Convolution Computation by TensorOps:

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.

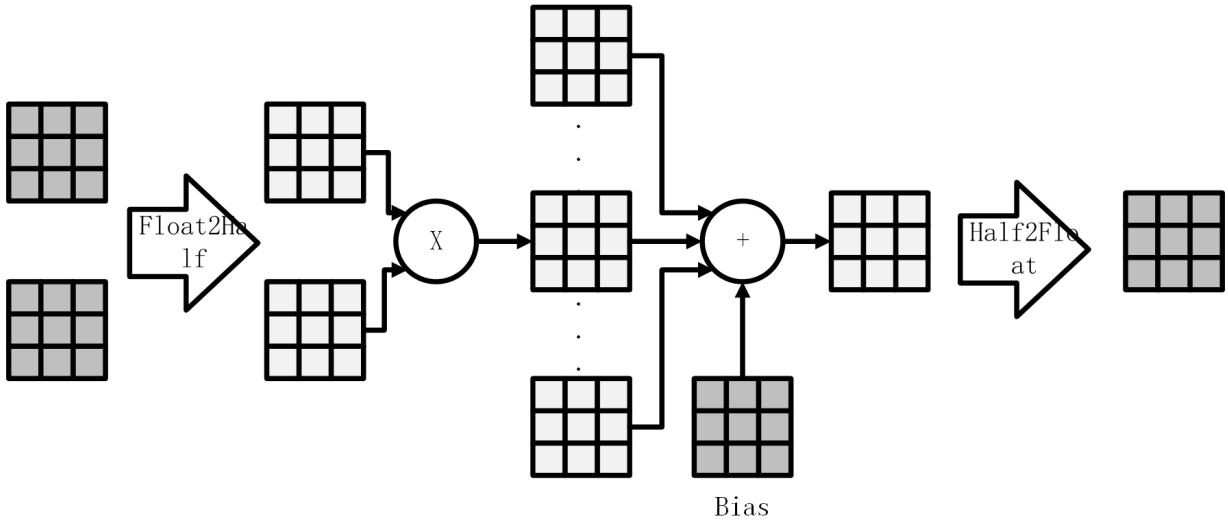


Figure 7.3: Brief illustration of NVIDIA's solutions [4]

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 converted to single-precision before accumulation explicitly. The high-level idea is

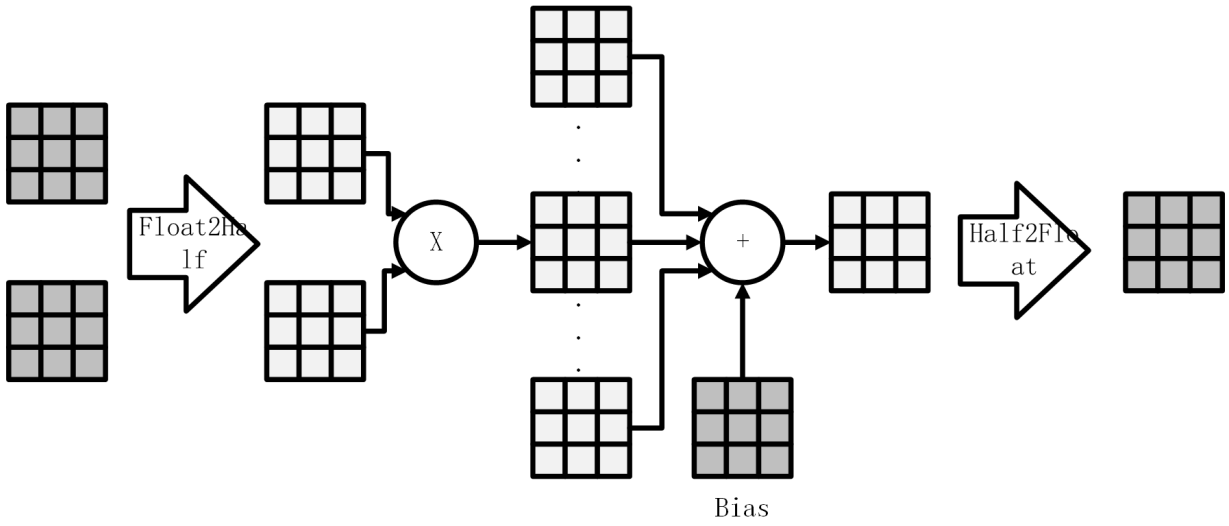


Figure 7.4: Brief illustration of our software solutions

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 previous 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 training 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 Resnet50, the

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]

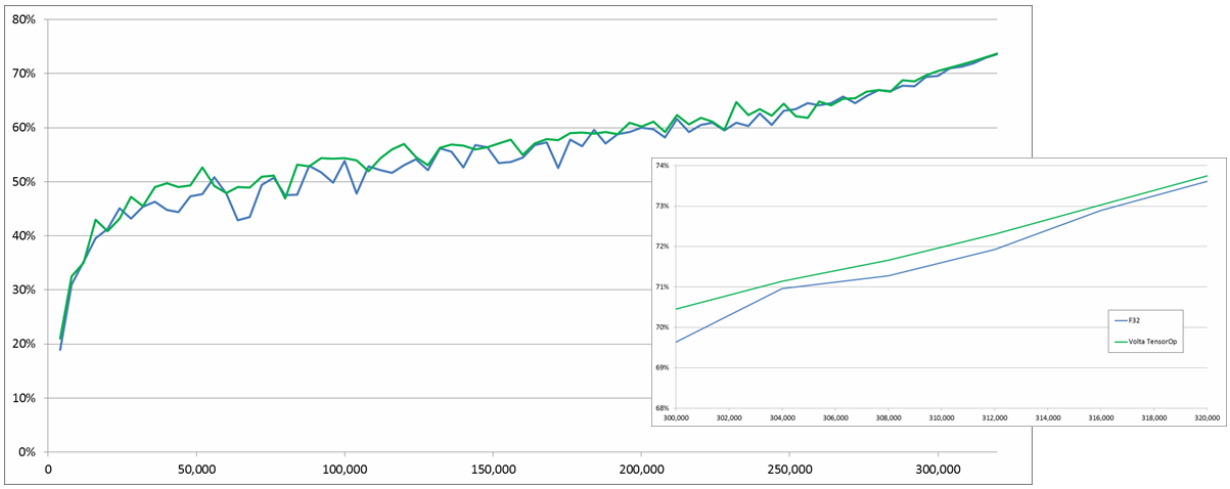


Figure 7.5: Accuracy figure of Resnet50 training under float16 and float32 [4]

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

Future Plan

8.1 Remaining Work

Currently, all these three optimization methods discussed in previous chapters are in prototype stage for pre-verification. The codes are quite dirty and unmaintainable, so refactoring is needed for further modifications. In addition to refactoring and finalized the code, several primary tasks are having been identified.

8.1.1 Energy Saving by Zero Copy

As mentioned in § V, compared to speed up, memory usage reduction and energy-saving are more critical benefits of this technique, as our framework executes on battery-powered devices. The energy usage reduction of different layers after optimization is going to be measured and reported.

8.1.2 Deployment of Convolution Neural Network Compression

Currently, the prototype of Convolution Neural Network Compression executes on desktop platforms. The deployment of the mobile platform is not a difficult task and is going to be finished at the beginning of the second semester. Meanwhile, the experiment of LeNet is also measured on desktop platforms. With the help of Zero-Copy, our algorithm will perform better, as the overhead of massive memory-copy is eliminated on mobile platforms.

8.1.3 More Experiments of Convolution Neural Network Compression

The reason for choosing LeNet in our experiments of neural network compression section is that LeNet is easy to train. To analyze the performance of our algorithms, a variety of convolution neural network is going to be benchmarked. We will focus on small convolution

neural network, while large ones will also be covered. Meanwhile, more detailed profile of memory behaviour is going to be given to further analyze the overhead.

8.1.4 Iterative K-means Clustering

The K-means clustering algorithm used in CPT is still a trivial implementation, which is time and space consuming. The training of LeNet with CPT took about two hours on our workstation, while five minutes is enough for the training without CPT. It is acceptable as a network only needs to be trained once. However, LeNet is quite small compare with convolution neural networks for complex tasks, such as ResNet on Cifar-1000, which typically needs about one week for training. The iterative K-means clustering method introduced in section 2.3 is going to be implemented.

8.1.5 Automatic deployment

In this prototype, the deployment of the compressed neural network still needs to be manually set. More specifically, after NNM finds the best clustering partition, the parameters need to be updated manually according to it, which is complicated, error-prone and time-consuming. In the rest of this project, an interface is going to be implemented, so that the compressed neural work can be deployed automatically.

8.1.6 Implementation of Half Precision

Half precision supporting is still under development. Both implementation and test are going to be finished in the second semester.

8.2 Future Plan

8.2.1 Training Stage optimization

Whereas MobileDL mainly focuses on the inference phase, an important avenue for future work is to optimize the local execution of the training stage. As mobile devices are usually used as data collectors, it is more natural to offload the training-phase locally. Furthermore, as data is produced so fast, pre-trained models may become out-of-date in few days. For example, for autopilot car, it needs to learn real-timely to adapt to rapid changed roads and traffic conditions. The focus should be on how to optimize fine-tuning method to reduce computation needed.

8.2.2 Discussion of IEEE 754

The range of float16 in IEEE 754 has a severe problem for convolutional neural networks training. The largest number of float16 in IEEE 754 is 65594 which is less than 256×256. As convolution neural networks usually deal with images, whose ranges are [0, 255] for all its three channels (RGB). This indicates that if the data is not normalized, any square operation together with scale operations may lead to Inf for float16. This kind of combination is quite common in many layers, such as LRN, BatchNorm, and MVN. Inf is very dangerous for deep learning as it makes the gradient unpredictable, the training can easily fail even only with one Inf or NaN. While if the images are normalized, that is, the range of all its three channels (RGB) are normalized to $[0, 1]$, Inf or NaN is less likely to appear. It seems that the problem is solved, however, we will suffer from another problem. According to Ginsburg's research [4], the distribution of numerical values in the training stage is shown in figure 8.1. It can be seen that around 25% of values are round to 0, which may cause gradient vanishing, while for a large interval of the range are unused. An important avenue for future work is to explore whether IEEE 754 standard is the best choice for deep learning, and if not, how these 16 bits should be divided to take the best advantage of them.

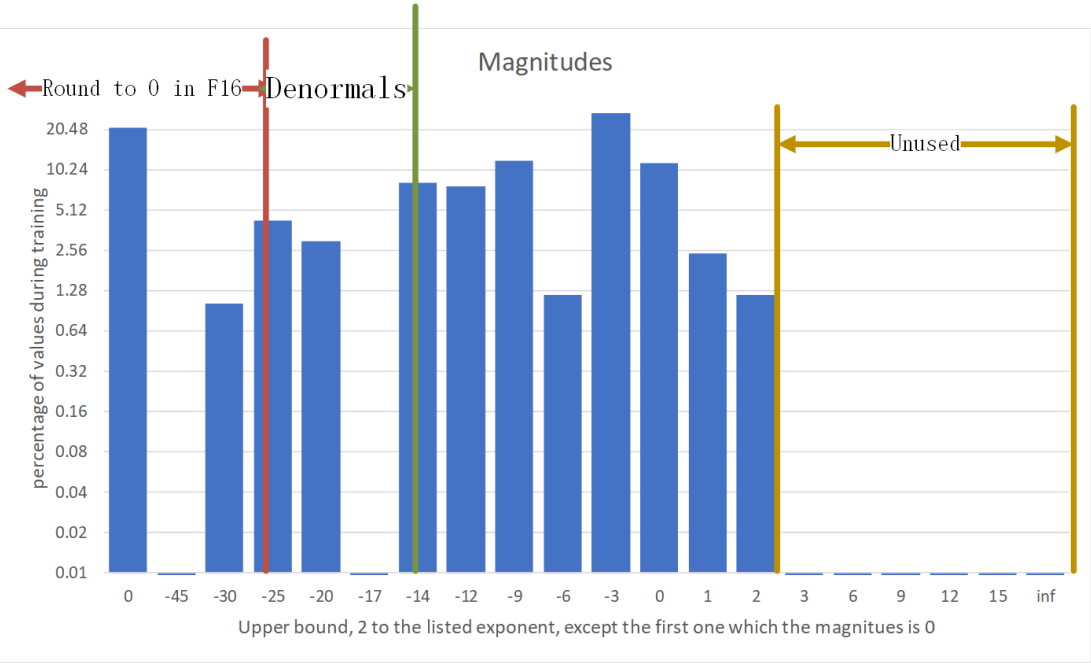


Figure 8.1: Distribution of numerical values in the training stage

Bibliography

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

[2] Mar 2017. [Online]. Available: https://developer.apple.com/library/content/documentation/3DDraw

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

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

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

[52] 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.

[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/