Optimization of Convolutional Neural Networks for Constrained Devices through Binarization

by

Kaveh Rouhandeh, BSc

A thesis submitted to the Faculty of Graduate and Postdoctoral Affairs in partial fulfillment of the requirements for the degree of

Master of Applied Science

in

Electrical and Computer Engineering

Carleton University
Ottawa, Ontario

Abstract

CNNs are the most common branch of Deep Neural Networks (DNNs), and they are structures with a strong capability for feature extraction. By using CNNs, a nonlinear model is trained to map an input space to a corresponding output space. These high-performance CNNs come with a high computational cost and the need for huge memory storage due to the chains of many Convolutional Layers (usually more than 50 layers). To address these issues, a variety of algorithms have been proposed in recent years. In this research, we present a solution that is a combination of several different approaches. and based on matrix optimization, parameters binary quantization, and data parallelism programming techniques. We show that our method significantly outperforms the current conventional PyTorch convolution operation with less memory usage and better computational budget when tested in different scenarios.

Acknowledgments

I would like to express my special thanks of gratitude to my supervisor Dr. Chris Joslin who gave me the golden opportunity to do this wonderful project which also helped me in doing a lot of research and I came to know about so many new things I am really thankful to him.

Secondly, I would also like to thank my family and my friends who helped me a lot in finalizing this project within the limited time frame.

Table of Contents

Terminology and Acronyms

FLOP Floating Point Operations per second

BOP Number of Bit-OPerations in a neural network

TOPS/W Tera Operations Per Second per Watt is a measure for energy efficiency which mostly used for evaluating the efficiency of AI hardware

Top-N Top-N is an Accuracy measure that takes the model predictions with higher probability. If one of them is a true label, it classifies the prediction as correct

AP Average Precision is a popular metric in measuring the accuracy of object detectors like Faster R-CNN, SSD, etc. Average precision computes the average precision value for recall values between 0 and 1

AlexNet AlexNet is a CNN that is 8 layers deep, from Krizhevsky et al [60].

ImageNet ImageNet is a large dataset of annotated photographs intended for computer vision research [61].

VGG-16 VGG-16 is a convolutional neural network that is 16 layers deep, from Hammad et al [62].

SSDnet The Single Shot multi-box Detector network (SSDnet) uses a single-stage object detection network that merges detections predicted from multiscale features, from Liu et al [76].

MNIST The Modified National Institute of Standard and Technology database is a large database of handwritten digits that contains 60,000 training and 10,000 testing images, from Yann et al [55].

ResNet Short for Residual Network is a specific type of neural network that was introduced in 2016. Different versions of Resnet have been developed. ResNet usually represents a number of its layers in form of ResNetN in which N is number of layers. For example, ResNet50 means represents a ResNet network which made of 50 layers, from Fung et al [63].

CIFAR-10 The CIFAR-10 dataset consists of 60000 32x32 color images in 10 classes, with 6000 images per class, from Krizhevsky et al [64].

Faster R-CNN Faster R-CNN is a deep convolutional network used for object detection, that appears to the user as a single, end-to-end, unified network, from Ren et al [65].

Mask-R-CNN Mask-R-CNN is a CNN and state-of-the-art in terms of image segmentation and instance segmentation. Mask R-CNN was developed on top of Faster R-CNN, from He et al [56].

Cooley-Tukey An Algorithm for the Machine Calculation of. Complex Fourier Series [66].

MobileNet MobileNet is one of the smallest Deep Neural networks that are fast and efficient and can be run on devices without high-end GPUs, from Howard et al [67].

GoogLeNet GoogLeNet is a 22-layer deep convolutional neural network that is a variant of the Inception Network, the network developed by researchers at Google, from Alake et al [68].

List of Tables

List of Figures

Chapter 1:

Introduction

Deep Neural Networks (DNNs) and Machine Learning (ML) algorithms are among the most powerful and successful techniques for image, video, and sound processing. We use DNNs in different fields of computer vision, including image classification, object detection, semantic segmentation, scene understanding, medical image analysis, etc. as they are able to distinguish the subtle nuances within the image in ways conventional algorithms cannot. Convolution Neural Networks (CNNs) are the most common branch of DNNs and are structured with a strong capability for feature extraction. Essentially a CNNs, is a nonlinear model trained to map an input space to a corresponding output space. These high-performance CNNs come with a large computational cost due to the chains of multiple Convolutional Layers, and the need for huge memory storage. In the other words, although CNNs are powerful and useful for media processing, they are both computation and memory intensive. Therefore, CNNs often require implementation on GPUs or highly optimized distributed CPU architectures to process large datasets, and such large CNNs also require a large amount of conventional memory.

1.1 Problem Statement

As it can be assumed, deploying a CNN on anything but the best hardware can pose a difficult problem, but there are situations that require the use of hardware that has limitations. One of those limitations is the use of low power (i.e., watts) devices that are designed to fit into spaces that have little to no cooling capacity. This means that as the power requirements are kept low, to avoid unnecessarily overheating the device and other components, the computing capacity is also reduced. This consumed power versus computing power relationship means that the device's computing capability is also reduced. Therefore, to do any kind of video processing in such a constrained device,

beyond current conventional algorithms means improving the performance of the CNN, and reducing the memory usage, so that it can work on such constrained devices.
For comparison and setup, our target device was an Nvidia Jetson Nano [13]. This Nano consumes a maximum of 12w (low compared to other Single Instruction, Multiple Data type devices), and has a maximum performance of 472 GFLOPS and an onboard memory of 2GB. The device is simply used to benchmark a comparison with other methods, the main idea is to improve the performance and memory usage. Overall, the conditions here are power limitation (10 W), memory limitation (2 GB), and no chance to upgrade the hardware. The development environment in this project is an NVIDIA Jetson Nano 2GB. This hardware has 128 GPU cores and 2GB of RAM, shared between CPU and GPU, and its CPU is Quad-core ARM® A57 @ 1.43 GHz.

1.2 Objective and Proposed Method

Therefore, our objective is to improve the execution efficiency and decrease hardware resource usage of a CNN
Our method focused on both memory utilization and process speed by combining matrix optimization techniques approach and parameter quantization.
We used Memory Efficient Convolution [10] as the main convolution function and combined this with a binarization method for parameter quantization. This was expanded with a training model for binary neural networks. In the training model, we proposed a transformation function, called SignShift, for transferring learned knowledge from a well-trained full precision network to its corresponding binarized network. Lastly, we provided an OpenCL-based implementation for our proposed method so that it would be easily transferable to other platforms (rather than being Nvidia specific)

1.3 Our Contribution

Our contribution to this research project can be categorized into three distinct aspects. The following list shows these aspects:

- **Research**

We focused on researching proposed methods and existing work to address the issue. We checked the convolutional operation optimization problem from various points of view.

- **The Method Development**

  We developed a parallel processing implementation for our proposed method, which is a combination of three different approaches to address our problem:

  - Used of MEC [10] and binarization together
  - Used an innovative training model for training a binary quantized network
  - Proposed a new method for transferring learned knowledge from a well-trained full precision network into its equivalent binary network

- **Evaluation**

  We defined different test case scenarios to evaluate the proposed method under various conditions.

1.4 Thesis Organization

The rest of this thesis is organized as follows:

- Chapter 2 discusses previous attempts at optimizing CNNs. In this section, we introduce different used methods to improve CNNs and convolutional operation.
- Chapter 3 covers our methodology including the proposed matrix multiplication algorithm, details of our strategies for using parameter quantization, and our learning approach in binarized CNNs.
- Chapter 4 outlines the advantages of our algorithm against current algorithms. We show the results of our method tested in different environments, especially NVIDIA Jetson Nano 2G.
- Chapter 5 is concluding the discussion and we provide an overall view of the method. Future work is discussed.

Chapter 2:

Literature Review

Owing to increased computing power and a sufficient amount of training data being available, DNNs have evolved into wider and deeper architectures. DNNs can have tens of thousands of layers, with billions of parameters [85]. Consequently, it has become increasingly challenging for researchers to deploy DNNs in portable devices with limited hardware resources (e.g., memory, bandwidth, and energy). Therefore, means are urgently sought for efficiently deploying DNNs in resource-constrained edge devices (e.g., mobile phones, embedded devices, smart wearable devices, robots, drones, etc.) [84]. Several optimization methods have been suggested to address this issue, these optimization methods reduce complexity and are also known as network compression techniques We divide these into four categories: network pruning, matrix optimization, transformation and hardware optimization, and parameter quantization. The following methods will be discussed in this chapter:

- Network Pruning
    - Weight pruning, Han et al. [6]
    - Channel pruning, Hao Li et al. [4]
    - Filter pruning, Wang et al. [2]
- Matrix Optimization
    - Image to Column, Chellapilla et al. [8]
    - Memory-efficient Convolution, Cho et al. [10]
    - Efficient matrix multiplication, Cong et al. [9]
    - Fast Fourier Transform, T. Abtahi et al. [12]
    - Overlap-and-Add convolution, Highlander et al. [14]
    - Split convolution, Abtahi et al. [13]
    - ZeroSkip-AddOpt, Park et al. [15]

- Transformation and Hardware Optimization
    - Transformation, Yous et al. [1]
    - Singular Value Decomposition Approximation-based solution, Chang et al. [7]
- Parameter Quantization
    - Fixed point quantization, Vanhoucke et al. [27]
    - Quantization Award Training, Krishnamoorthi [28]
    - Bi-Real, Yang et al. [18]
    - BinaryConnect, Bengio et al. [19]
    - XNOR-Net, Rastegari et al. [32]
    - Accurate Binary Convolutional Neural Network, Lin et al. [37]

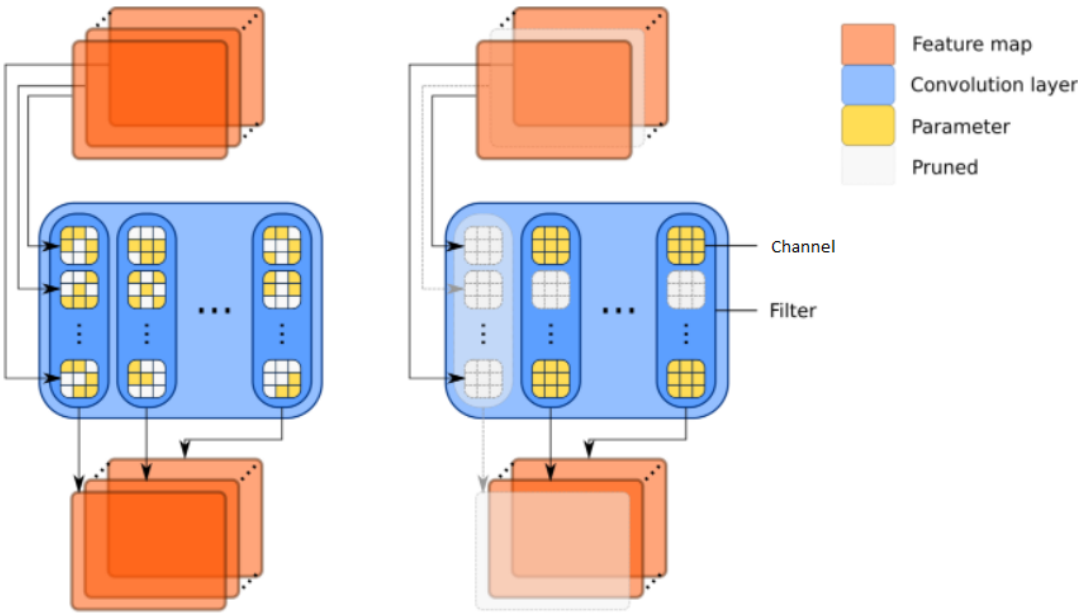
2.1 Network Pruning

CNN pruning has become one of the most common and successful strategies for network compression. The idea of pruning is based on removing unnecessary parameters in a network. Having fewer parameters means less computational effort. Pruning is conducted on different levels, including connections, nodes, channels, and filters. Generally, network pruning can be categorized into unstructured (weight) pruning and structured pruning [3]. The initial step to applying the pruning methods is training. In fact, pruning is applied to well-trained networks, and it cuts the least important parameters in the pre-trained network. Weight pruning zeros out specific weights in filters which means that the size of filters does not change after applying this method. To satisfy this approach, we need to make hardware and software implementation changes. As the network structure remains the same, so it is called unstructured pruning. On the other hand, structured pruning makes changes to the structure of the convolutional layer by changing input and output shape (Figure 2.1). As shown in Figure 2.1.a, in unstructured pruning only some parameters of each channel are pruned, and the number of multiplication operations decreases which has no effect on the input and output matrix size. . On the other hand, in structured pruning shape (size) of input and output matrices are changed. By pruning a filter, the channel size of the output matrix is decreased. Also, by channel pruning a channel of the input matrix is ignored. Figure 2.1 provides a visual comparison between

all different types of pruning [5] [59]. Figure 2.1.b illustrates this pruning. As discussed, filter pruning, and channel pruning change the input and output metrics' shape. These changes have direct effects on other layers in the network therefore they are known as structured pruning.

The following is the general categorization of pruning strategies:

- Unstructured Pruning
    - Filter Shape Pruning (Weight Pruning)
- Structured Pruning
    - Channel Pruning
    - Filter Pruning



(a) Unstructured Pruning **(b) Structured Pruning**

Figure 2.1 Unstructured Pruning vs Structured Pruning [59]

2.1.1 Filter Shape Pruning (Weight Pruning)

Han et al. [6] are the first researchers who presented the DNN weight pruning methods. They pruned the weights with small magnitudes and retrained the network model both heuristically and iteratively. As illustrated in Figure 2.2, their proposed method employs

a three-step process that begins by learning connectivity via regular network training. Unlike conventional training, instead of learning the final values of the weights, the method learns important connections in the network. In the second step, the low-weight connections with weights below a threshold are pruned. The last step retrains the network from the final weights for the remaining spare connections. Also, they concluded that without the third step, the accuracy of the network would experience a sharp reduction and the third step helps keep the remaining connection as strong as possible. Han et al. [6] showed that by applying this method, the number of parameters of AlexNet [60] on the ImageNet [61] dataset can be reduced by a factor of 9-times, from 61 million to 6.7 million, without a sharp change of accuracy.

A similar experiment with VGG-16 [62] showed a 13-times reduction in the number of parameters from 138 million to 10.3 million again without incurring accuracy.

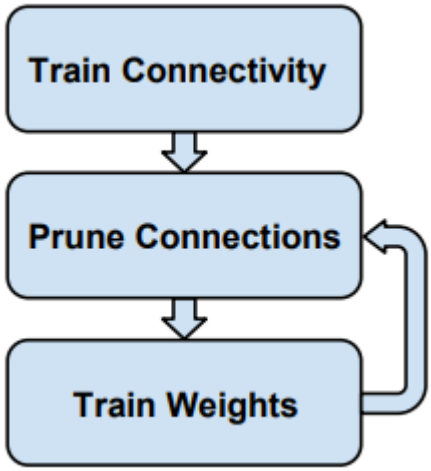


Figure 2.2 Three-Step Training Pipeline [6]

2.1.2 Channel Pruning

Hao Li et al. [4] showed that by applying zero as a threshold on the SSDnet [76], which is trained by a custom dataset, almost 76% of the CNN channels can be removed, and still the same AP can be reached. Table 2.1 and its corresponding chart (Figure 2.3) show Hao Li al. [4] experiments' results of different scenarios of channel pruning on the same pre-trained network. The method used by Hao Li et al. [4] was based on removing the channel of a filter that has the closest weight to zero in a convolution layer. Furthermore, Hao Li al [4] showed that inference costs for VGG-16 [62] and ResNet-110 [63] can be

decreased up to 34% and 38%, respectively, on the CIFAR10 [64] dataset by applying simple channel pruning. Inference in DNNs refers to the process of using a trained DNN model to make predictions against previously unseen data**.** They also showed that this method could achieve around a 30% reduction in FLOP for VGG-16 [62] and deep ResNet-110 [63] on CIFAR-10 [64] without significant loss in the original accuracy. The metric used in Hao Li al [4]'s work to show the accuracy is AP which is a popular metric in measuring the accuracy of object detectors like Faster R-CNN [65], SSDnet [76], etc. Average precision computes the average precision value for recall values over 0 to 1.

Table 2.1 Channel Pruning on a Pre-trained SSDnet's Accuracy

Approximate percentage of CNN channels left	**Total parameter count**	**AP**
100%	3,333,120	0.500
96%	3,238,764	0.502
83%	2,945836	0.505
60%	2,410,124	0.501
26%	1,596,297	0.502
24%	1,569,638	0.500
23%	1,536,326	0.497
19%	1,453,190	0.333
17%	1,396,832	0.025
10%	1,234,942	0.001

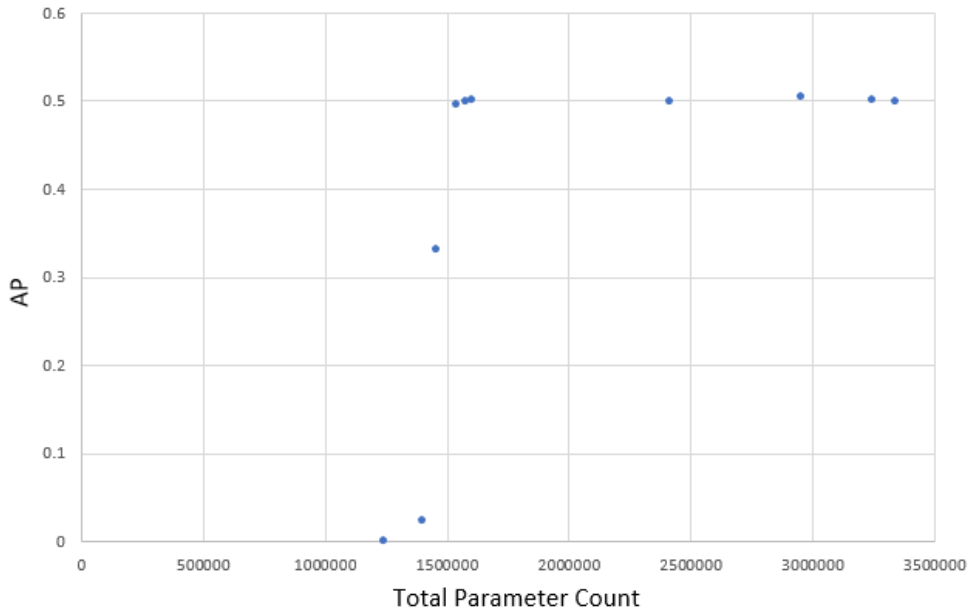


Figure 2.3 Channel Pruning on a Pre-trained SSDnet's Accuracy [4]

2.1.3 Filter Pruning

Wang et al. [2] used statistical modeling to measure the redundancy in each convolutional layer, showing that pruning filters in the layer with the most redundancy outperform pruning the least important filters across all layers. They also pointed out that by least important filters, they mean filters which values are almost zero and have no effects on the output matrix. Wang et al. [2] showed that by applying this method on ResNet-20 [63] and ResNet-56 [63] on CIFAR-10 [64], the number of floating-point operations experienced a 45.8% and 53.8% reduction, respectively, and accuracy even increased slightly.

The main challenge in pruning strategy is finding a threshold value that helps distinguish unnecessary parameters from others. Depending on the network architecture and training data, this value will be different. This means that for each network architecture and each training dataset, we need to perform a new experiment to find the best-fit pruning method and its corresponding hyperparameters.

On the other hand, implementations of the pruning methods are not fully hardware friendly. Ma et al. [5] addressed this issue and mentioned that unstructured pruning is not suitable on GPU and multi-core CPU, whereas structured pruning is suitable on both hardware types. They mentioned that this phenomenon happened because we need to use

extra memory for storing indexes of parameters that are pruned. Also, more computation is required to check whether a parameter is pruned or not.

2.2 Matrix Optimization

For matrix convolution optimization problems, part of the solution is related to the mathematical aspect of convolution operation and the benefits we can bring to the solutions using a mathematical approach. Figure 2.4 illustrates an overview of convolution operation for two-dimensional input and filter. The most straightforward implementation of convolution operation, called direct implementation, uses three nested loops (a loop per each dimension). If we have matrix I as an input of size $W_i \times H_i \times C_i$ and F as a filter matrix of size $W_f \times H_f \times C_i \times C_f$, the complexity of the basic implementation would be $O(n^3)$. Although direct implementation is not an optimal way to perform convolution operation, it has no memory overhead. So, it could be one of the possible solutions when the restriction is memory capacity. When working with multicore processors like Field Programmable Gate Arrays (FPGA) and Graphics Processing Units (GPU), we can use task parallelism to compute the convolution operation. In data parallelism, multiple cores are used simultaneously, and data is shared between the cores. Consequently, parts of redundant computations and nested loops are avoided by parallelism computing.

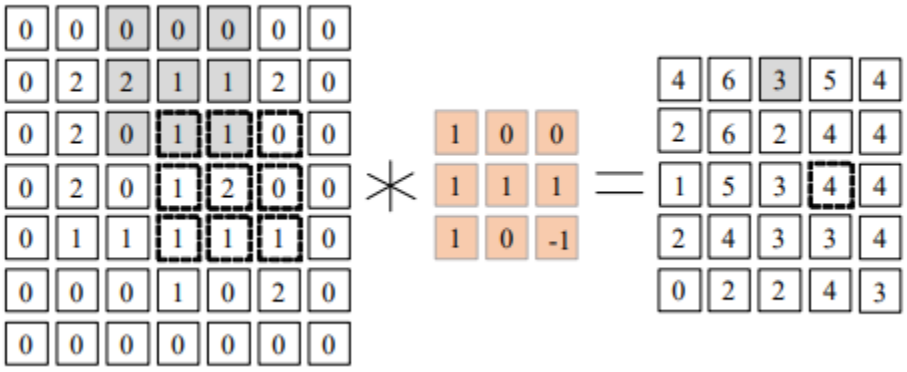


Figure 2.4 Illustration of Two-Dimensional Convolution Operation [10]

2.2.1 Image to Column (im2col)

General Matrix Multiply (GEMM) [52] is a common matrix multiplication algorithm in linear algebra, machine learning, statistics, and many other domains. For any matrices A and B of size $m \times k$ and $k \times n$ in turn, C is a matrix of size $m \times n$ and $C = A \times B$. C is calculated using two nested loops over A and B elements in the direct implementation. Using the GEMM, we can use the parallelism computing concept to compute the same output.

Chellapilla et al. [8] proposed a GEMM-based algorithm which is called Image to Column (im2col). This algorithm is used to convert convolution operations into matrix multiplications. The first step is to turn the input from a 3D matrix into a 2D matrix then we can treat it like a matrix. Each kernel is applied on a small three-dimensional cube within the image, so each of those cubes of input values is taken, and they are copied out as a single column into the matrix which is named as the lowered matrix (Figure 2.5). The same method is applied to the filters. By the end of this step, we will have two 2D matrices Input matrix of size: (output_size $\times$ a_filter_size) and the filter matrix of size: (a_filter_size $\times$ number_of_filters).

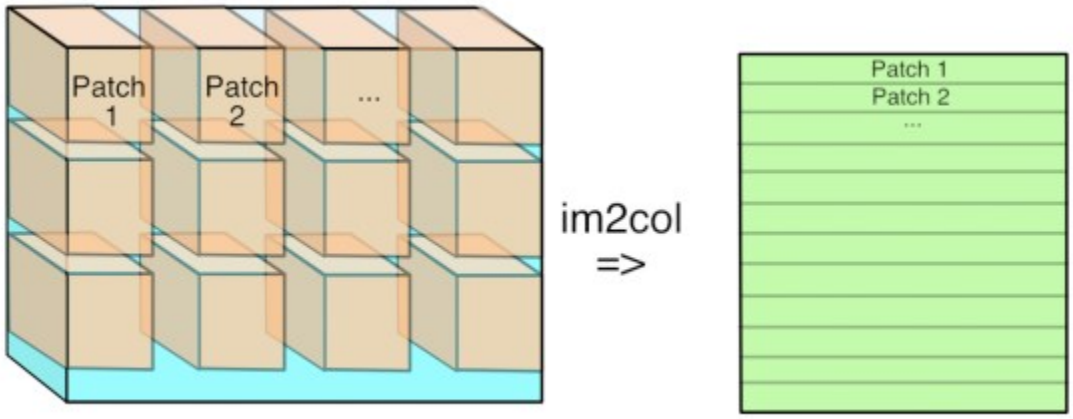


Figure 2.5 im2col the Lowered Matrix [52]

In Figure 2.5, the green matrix is the lowered matrix. Although the im2col method brings the advantage of parallel computing to the solution, it has memory overhead of size image-matrix plus kernel-matrix; this is because of the significant amounts of redundancy in the lowered matrix, especially when the stride is a small number, and the filter size is

large. The Stride is how far the filter moves in every step along one direction. This means that pixels included in overlapping filter sites will be duplicated in the image matrix, which is inefficient. Additionally, the overhead becomes even worse when the filter size is relatively much smaller than the input matrix.

2.2.2 Memory Efficient Convolution (MEC)

Cho et al. [10] proposed a memory-efficient convolution (MEC) algorithm. MEC can reduce either memory consumption or memory-bus traffic (less traffic from global memory to shared memory) on the GPU [8] and further improve the performance of computing convolution operations. As with the im2col-based algorithm, MEC is based on creating the lowered matrix. The primary motivation of MEC is to reduce the amount of redundancy in the lowered matrix and keep the computation pattern GEMM-compatible. To address this issue, MEC uses multiple columns lowering at once rather than every individual patch. Figure 2.6 illustrates the idea and details of MEC with a simple example. In this example, MEC copies matrix W (shaded in Figure 2.6) of size $H_i \times W_f$ into one row of matrix L. In this example, L is the lowered matrix, and each of its rows is a copy of a block of size $H_i \times W_f$ from the input image. In the same way, to generate the next row of the matrix L, we just need to shift W by value of stride. Once the lowered matrix L is formed, MEC multiplies L by F. For performing the multiplication, MEC creates another set of vertical partitions (which are $\{P, Q, R, S, T\}$ in Figure 2.6) within L. Each partition is of size $W_o \times W_f \times H_f$ and each partition is obtained by shifting to the right by SW_f which S is the size of the stride. Eventually, each row of the output matrix is computed by multiplying one of the mentioned partitions ($\{P, Q, R, S, T\}$) and F. in this example, MEC could reduce memory overhead from 25×9 (in the im2col-based algorithm) to 5×21.

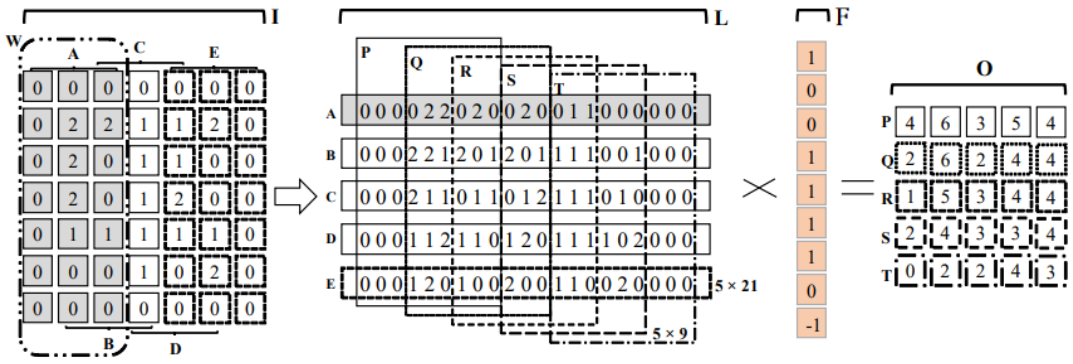


Figure 2.6 MEC Example for the Same Problem in Figure 2.4 [10]

Cho et al. [10] showed that in comparison to the im2col-based algorithm, an MEC CPU-based implementation could reduce the memory-overhead 3-times and improve runtime by almost 20% for a large-scale convolutional DNN like ResNet-110.

Another orthogonal direction to enhance the performance of a convolution operation is to reduce the theoretical number of basic operations needed in the convolution operation.

2.2.3 Minimizing Multiplication

Cong et al. [9] proposed an algorithm for matrix multiplication that decreases the number of multiplication operations. In this work, the computation in the convolution layers of a CNN is expressed in a convolutional matrix multiplication representation. In the new representation, the input image is reformed into a vertical vector of size: $H_i W_i \times 1$, and filters are represented as a matrix of size: $H_i W_i \times H_o W_o$, and the output will be a vertical vector of size $H_o W_o \times 1$consequently. In this representation, the outcome will be calculated by $O = F \times I$. After forming the convolution operation into a matrix multiplication problem, Cong et al. [9] proposed an efficient way to compute two matrix multiplications. They clarified the idea using an example of the multiplication of two 2×2 matrices: suppose we have two 2×2 matrices by using direct matrix multiplication. In that case, we need to do eight multiplications and four additions (Equation 2.1). In contrast, by using extra computation, we can define M values like Equation 2.2 and by using M values, calculate the output as it is shown in Equation 2.3.

$$W = \begin{pmatrix} W_{1,1} & W_{1,2} \\ W_{2,1} & W_{2,2} \end{pmatrix}, X = \begin{pmatrix} X_{1,1} & X_{1,2} \\ X_{2,1} & X_{2,2} \end{pmatrix}, Y = \begin{pmatrix} Y_{1,1} & Y_{1,2} \\ Y_{2,1} & Y_{2,2} \end{pmatrix}$$

$$\begin{aligned} Y_{1,1} &= W_{1,1} \times X_{1,1} + W_{1,2} \times X_{2,1} \\ Y_{1,2} &= W_{1,1} \times X_{1,2} + W_{1,2} \times X_{2,2} \\ Y_{2,1} &= W_{2,1} \times X_{1,1} + W_{2,2} \times X_{2,1} \\ Y_{2,2} &= W_{2,1} \times X_{1,2} + W_{2,2} \times X_{2,2} \end{aligned} \quad \textbf{(2.1)}$$

$$\begin{aligned} M_1 &= (W_{1,1}+W_{2,2}) \times (X_{1,1}+X_{2,2}) \\ M_2 &= (W_{2,1}+W_{2,2}) \times X_{1,1} \\ M_3 &= W_{1,1} \times (X_{1,2}- X_{2,2}) \\ M_4 &= W_{2,2} \times (X_{1,1}- X_{2,2}) \\ M_5 &= (W_{1,1}+W_{1,2}) \times X_{2,2} \\ M_6 &= (W_{2,1}- W_{1,1}) \times (X_{1,1}+X_{1,2}) \\ M_7 &= (W_{1,2}- W_{2,2}) \times (X_{2,1}+X_{2,2}) \end{aligned} \quad \textbf{(2.2)}$$

$$\begin{aligned} Y_{1,1} &= M_1+ M_4 - M_5+ M_7 \\ Y_{1,2} &= M_3 + M_5 \\ Y_{2,1} &= M_2+ M_4 \\ Y_{2,2} &= M_1- M_2 + M_3+ M_6 \end{aligned} \quad \textbf{(2.3)}$$

The algorithm proposed by Cong et al. [9] experienced up to 47% reduction in computation for a CNN used in the 2012 ImageNet contest [11]. Depending on the target hardware and the size of the matrices, the overhead of more addition operations could eliminate the benefits of the multiplication operations that were saved compared to normal matrix multiplications. In this simple example, although the number of the applied multiplications decreased from 8 to 7 without changing the computation results, the number of additions increased from 4 to 18. Also, this algorithm did not deal with memory overhead for reforming input image and filter matrices.

2.2.4 Fast Fourier Transform-based Algorithms

Another set of solutions relies on the fact that convolution operation can be done as simple multiplication in the frequency domain. These algorithms are called Fast Fourier Transform-based (FFT) algorithms. Two different approaches are proposed in CNN processing in the FFT domain, which implement networks in the training and inference phases. In both the training and inference phases, FFT is applied to decrease the number of computations of the CNN [12]. Suppose we have input matrix I as a square matrix of size $N \times N$ and a simple square filter F of size $K \times K$. Equation 2.4 represents the convolution operation, which $\otimes$ is a notation of convolution operation. Using FFT, we can rewrite Equation 2.4 as Equation 2.5**,** where $\odot$ represents a Hadamard product.

$$O = I \otimes \mathrm{F} \tag{2.4}$$

$$O = FFT(I) \odot FFT(Padding(F)) \tag{2.5}$$

Padding is required for matrix F with some values (mostly zero) to make the size of F equal to the size of I. FFT can be calculated based on the Cooley-Tukey algorithm [66] with a cost of: $N^2 \log(N)$, and the $N^2$is the cost of the Hadamard process. We can conclude that the total cost of the basic FFT-based algorithm is: $N^2 \log(N)$, which is by far better than the traditional convolution operation. The main issue here is that in almost every convolution operation in CNN has $K \ll N$, which means that we need to add lots of padding when applying the FFT-based algorithm for computing the convolution of I and F. This redundancy is addressed by Highlander et al. [14] who proposed a method called Overlap-and-Add convolution (OaAconv). As illustrated in Figure 2.7**,** the input image is split into patches of the same size as the filter in this method. FFT operations are conducted on these small patches. After applying element-wise multiplication, the result will be calculated by reverse transferring to the spatial domain using an overlap addition. Although OaAconv could improve the functionality of basic FFT-based algorithms (the overall complexity in this method is improved to $N^2 \log(K)$), it still has redundancy issues. In fact, as illustrated in Figure 2.7**,** redundancy can be observed during overlapped addition, which is conducted with a number comparable to the image's patches. Therefore, the overlapped additions require many operations.

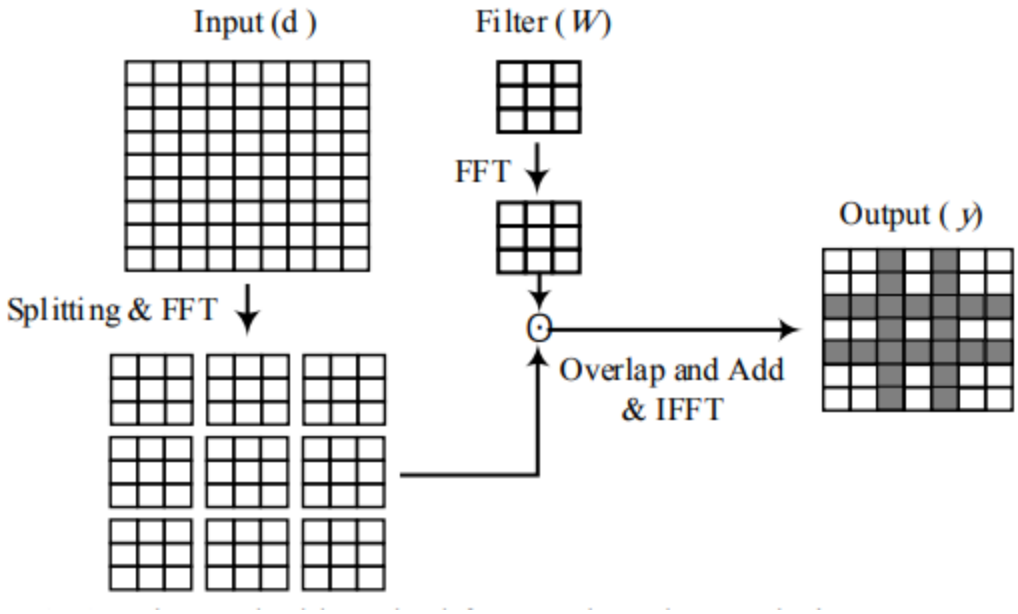


Figure 2.7 Illustration of OaAconv [14]

To solve this issue, Abtahi et al. [12] proposed a method that stands somewhere between basic FFT and OaAconv. In the proposed method, which is called SplitConv, the input image is split into patches with the size $S \times S$ which $K \leq S$. After splitting the input image, this method will apply the basic FFT-based algorithm on each patch and each filter matrix, which means that if we have input matrix I as a square matrix of size $N \times N$ and a simple square filter F of size $K \times K$ and S as patch size. The input matrix will be split into $\frac{N^2}{S^2}$ different patches. In the last step, FFT is applied on each patch and filtered by adding padding to the filters to make them the same size as the patches. Therefore, Equation 2.4 can be rewritten as Equation 2.6:

$$I = \{i_1, i_2, \dots, i_d\} \; and \; d = \frac{N^2}{S^2} \tag{2.6}$$

$$O = I \otimes \mathrm{F} = \{ i_1 \otimes \mathrm{F}, i_2 \otimes \mathrm{F}, \dots, i_d \otimes \mathrm{F}\} \tag{2.7}$$

Figure 2.8 provides an illustration of how the splitting is used in FFT-based processing. In this method, the complexity and memory overhead are dependent on the patch size. If we consider $S = K$, Splitconv will act like OaAconv, and if $K = N$, it will behave like the basic FFT-based algorithm. Abtahi et al. [12] showed that the complexity of

Splitconv is $N^2 \log(S)$. Another approach to improve the efficiency of convolution operation is using the Coppersmith-Winograd algorithm [16]. The Coppersmith-Winograd algorithm focuses on finding the minimum number of arithmetic operations needed to perform the computation. Using this algorithm, the complexity of the multiplication of two $n \times n$ matrices is $O(n^{2.37})$ which is obviously significantly smaller than the $O(n^3)$ [16].

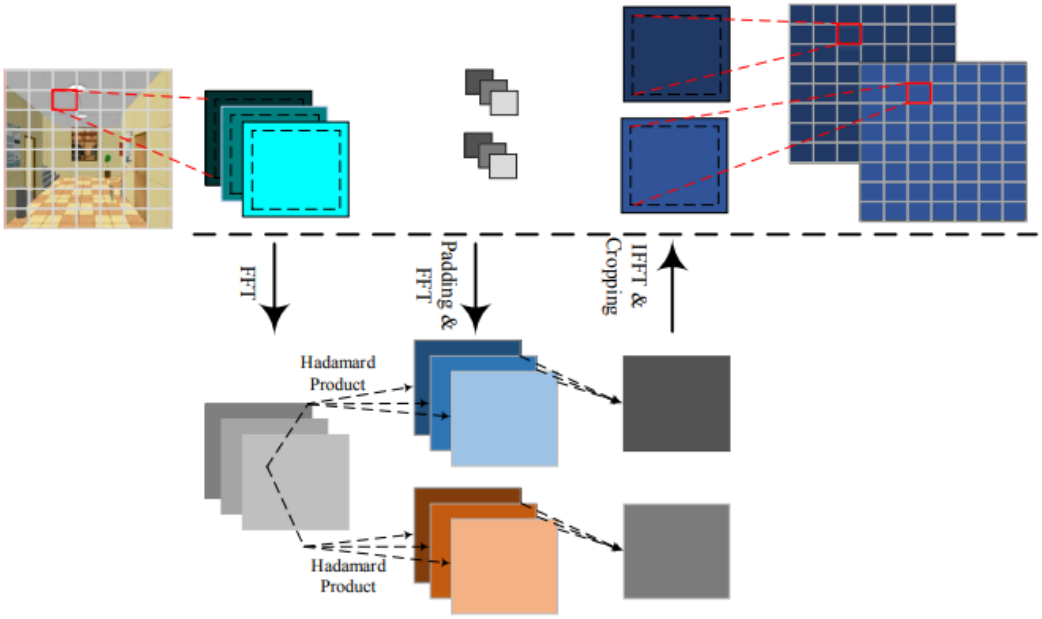


Figure 2.8 Splitconv Procedure [12]

The Coppersmith-Winograd algorithm is frequently used as a building block in other algorithms to prove theoretical time bounds [15]. Park et al. [15] proposed an algorithm for computing convolution based on the Coppersmith-Winograd algorithm. The Proposed algorithm [15] is motivated to address two issues. First, CNNs often exhibit a significant portion of zero filters; even pruning techniques have been applied [6]. This led them to skip multiplications and associated memory loads for zero results in Winograd convolution. They also proposed to detect zero input data at runtime and cut the execution of associated load and multiply instruction. Also, a small hardware unit was added inside the GPU architecture to perform zero checks and instruction flow control. The second fact is that Winograd's convolution's benefit can be limited by increasing the number of additions. Although Winograd convolution needs less multiplication, as the

input matrix size increases, it requires more additions to compute the convolution. The proposed solution for this issue is to exploit the data access behavior to reduce the algorithm's execution cycle of additional steps. Park et al. [15] called this algorithm ZeroSkip_AddOpp, which is a combination of presented solutions. ZeroSkip_AddOpp achieved up to 51.8% higher performance than the direct convolution.
Although FFT-based and Winogard-based algorithms could improve the runtime of a convolution operation compared to direct convolution, they are still slower than MEC and im2col-based algorithms. Cho et al. [10] experimented with different input matrix sizes and filter sizes. This study shows that regardless of input matrix size and filter matrix size, MEC is faster and more memory efficient than FFT-based algorithms and im2col-based algorithms.

2.3 Transformation and Hardware Optimization

Transformation and hardware optimization mostly focus on the implementation aspect of CNN-based algorithms that involve software or hardware architectural solutions. There is no tangible difference between transformation and hardware optimization. In other words, it is difficult to distinguish which solution belongs to which solution set. The transformation approach usually follows hardware optimization. Transformation techniques primarily make use of software implementation solutions to improve performance. Simple steps can be taken improve the performance of a CNN performance, such as removing redundant loops and using parallel computing techniques.
Yous et al. [1] showed that we could improve memory usage by up to 54% by focusing on implementation. Yous et al. [1] proposed an efficient implementation for the Winograd convolution [16] on in Intel Movidius Myriad 2 platform [73] that provide a set of vector processing units that access high bandwidth local memory. Chang et al. [7] have suggested a Singular Value Decomposition Approximation-based (SVDA) solution for reducing resource usage of two-dimensional convolution operations in CNNs. They showed that by their SVDA hardware implementation can achieve a reduction in resource usage by between 14.46% and 37.8%, while the loss of classification accuracy is less than 1%. The proposed SVDA decomposes the 2D convolution to pairs of low complexity 1D convolution operations by applying low rank approximation. Using a low-

rank matrix approximation to the kernel means that only some of the most significant singular values are kept, while the others are set to zero, which also provides an opportunity to decrease computation complexity again. Thus, the original 2D convolution is decomposed into m pairs of 1D convolution. In terms of complexity, the SVDA improves the complexity of the original 2D convolution from $O(n^3)$ to $O(2mn^2)$. Therefore, the complexity is reduced if $m < n/2$. The value of m is hyperparameter and should be decided by a trade-off between complexity and precision.

2.4 Parameter Quantization

Quantization methods try to reduce memory usage and the computation process without removing the redundant parameters. The main idea behind these methods is to use different data types instead of floating-point. Although we note that 32-bit floating-point arithmetic operations have been massively optimized in hardware like CPUs and GPUs, quantizing to other data types like integer precision removes the need for floating-point precision hardware and reduces energy consumption. Quantization can be applied in different levels of CNNs depending on the network architecture, the purpose of the neural network, and even the training dataset, etc. The same reasons have led researchers to develop a variety of quantization methods. Rishabh et al [17] proposed quantization techniques categorized into two general groups: Quantized training and post-training. Quantized training has been a popular approach focused on training neural networks using low-precision number representation for weights, activation, and gradients. This approach improves memory usage and computational complexity and decreases the training time of the network. Methods that use this approach replace floating-point operations and parameters with fixed-point data types, integers, etc., and then apply learning algorithms. In contrast, post-training strategies train the network with floating-point prepositions and will use quantization algorithms after the training process ends. The quantization methods can be categorized on another basis as well. In the new classification, the main concern is where the quantization is applied. Three possible parameters to apply quantization are weights, activations, and gradients. Based on this point, there are three different categories [33]:

- **Weight Quantization**

The motivation to quantize weights is to reduce the model size and accelerate the training and inference process. Shuang et al. [34] showed that quantized weights make neural networks harder to converge, and a lower learning rate is needed to ensure the network has good performance.

- **Activation Quantization**

Quantized activation replaces floating-point inner-products with faster operations like binary operations, which can further speed up the network training. Cai et al. [35] worked on activation quantization. They mentioned that one of the reasons that make the quantization of activation more difficult than that of weights is the need for back-propagation through the non-differentiable operators. Furthermore, Lin and Talathi [36] showed that the quantized activations could lead to "gradient mismatch" problems. This means that there is a discrepancy between the quantized activation and the computed backward gradient [33].

- **Gradient Quantization**

The motivation to quantize gradient descent is to reduce the cost of communicating the gradient updates between nodes during training process of large neural networks. To quantize the gradient, we must address how to take both the magnitude and sign of gradients into account since both factors are essential for updating the weights [33]. Furthermore, quantization algorithms can be categorized into different groups based on the quantization data type. Therefore, three approaches have been proposed: Fixed point quantization, Integer quantization and Binary Quantization.

2.4.1 Fixed Point and Integer Quantization

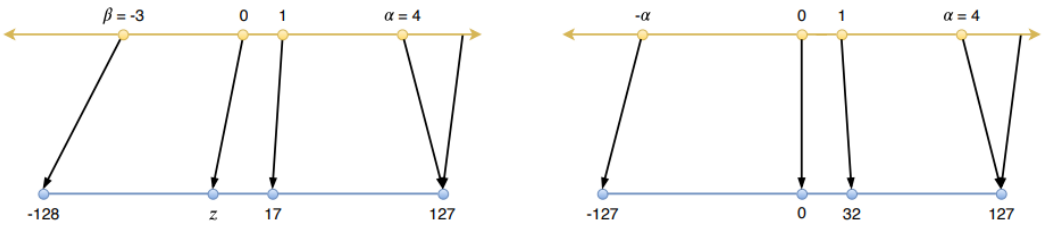
Vanhoucke et al. [27] showed neural networks could be quantized after training to use int8 instruction without any large reduction in the accuracy of the network. Also, it has been shown that in some cases where the network is based on a complex network architecture such as Mask-R-CNN, and the training dataset is not big enough, for maintaining the accuracy of the prediction, we may need to retrain the model when quantized for int8. Krishnamoorthi [28] called it Quantization Award Training (QAT), they evaluated several quantization methods and bit widths on a variety of CNNs. To

show that networks like MobileNet [67] do not reach baseline accuracy with int8 Post Training Quantization (PTQ) and require QAT even with per-channel quantization. To explain this, let [β, α] be the range of representable real values, and b is the bit-width of signed integer representation which is 8 in this case. The integer quantizer tries to map every weight value in the range: [β, α] to its corresponding signed integer representation in the range: $[-2^{b-1}, 2^{b-1} - 1]$. There are two choices available to transfer real values to integers: Affine Quantization maps and Scale Quantization maps. A parameter z is defined in Affine Quantization maps to determine the corresponding zero in the integer range. This mapper uses: $f(x) = s.x + z$ as a transformation function where s is the scale factor, and z is the zero-point, the integer value to which the real value zero is mapped [31] [27]. Equations 2.7 define s and z in affine transformation function:

$$s = \frac{2^b - 1}{\alpha - \beta} \tag{2.7}$$

$$z = -round(\beta.s) - 2^{b-1}$$

In Scale Quantization mapping, range mapping is performed with only a scale transformation. Figure 2.9 illustrates the mapping of real values to int8 with both Affine Quantization map and Scale Quantization map.



(a) Affine quantization (b) Scale quantization

Figure 2.9 Quantization Mapping of Real Values to int8 [28]

Vanhoucke et al. [27] evaluated int8 quantization on a variety of neural network tasks and models, summarized in Table 2.2. For evaluating the accuracy of the networks and models, Vanhoucke et al. [27] used a set of metrics based on the tasks. Metrics for all networks are reported as a percentage. In this experiment, the metric that was used for accuracy evaluation is based on top-n. Top-N is an Accuracy measure that takes the

model predictions with higher probability. If one of them is a true label, it classifies the prediction as correct. In the case of the top-1 score, we check if the top class (the one with the highest probability) is the same as the target label. As shown in Table 2.2, int8 quantization is not a good choice for detection, and this is the main reason we did not choose it over other methods.

Table 2.2 Summary of int8 Quantized Networks Accuracy

Task	Model	Accuracy (%)	Metric	Dataset
Classification	MobileNet v1	71.88	Top 1	ImageNet 2012
Classification	MobileNet v2	71.88	Top 1	ImageNet 2012
Classification	ResNetXt 50	77.61	Top 1	ImageNet 2012
Classification	ResNetXt 101	79.30	Top 1	ImageNet 2012
Detection	Faster R-CNN	36.95	AP	COCO 2017 [51]
Detection	Mask R-CNN	37.89	AP	COCO 2017

2.4.2 Binary Quantization

Compared with the full precision neural networks, the binary neural networks based on 1-bit representation replace the floating-point multiplication and addition operation with the efficient XNOR-Bitcount operations, thus greatly decreasing the storage usage and inference time. Figure 2.10 illustrates a binary convolution operation that works with XNOR and bit count instead of multiplication and addition operations. In binary quantization, quantized values are assigned with values -1 and +1 as binary “values” and their encoding, 0 and 1, as binary “encoding”. In Figure 2.10, corresponding chunks of the filter and the input (green boxes) are shown as two horizontal matrices. For operating the convolution operation using XNOR operation, BNNs use the truth table presented in Table 2.3.

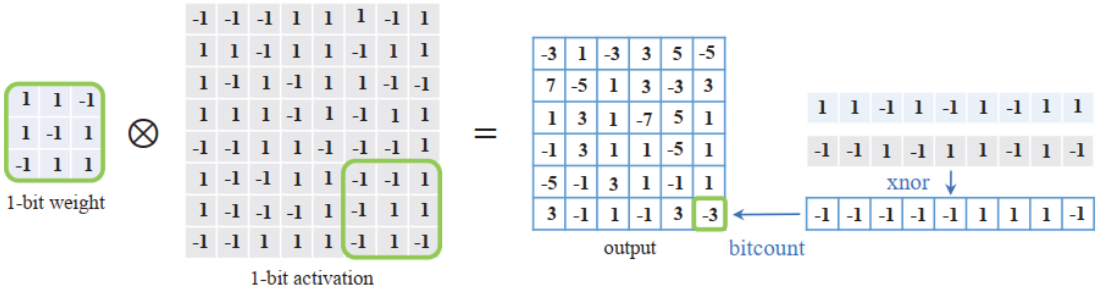


Figure 2.10 Convolution Process of Binary Neural Networks [20]

Table 2.3 XNOR Operation Truth Table

Encoding (Value)	Encoding (Value)	XNOR(Multiply)
0(-1)	0(-1)	1(+1)
0(-1)	1(+1)	0(-1)
1(+1)	0(-1)	0(-1)
1(+1)	1(+1)	1(+1)

Binarization can be applied in both forward and backward propagation. However, the binarization function is not normally differentiable, and even worse, the derivative value in part of the function vanishes (e.g., 0 for sign function) [20]. Therefore, the common gradient descent-based backpropagation algorithm cannot be directly applied to update binary weights [20]. To address the gradient problem occurring when training deep networks binarized by sign function, different techniques have been proposed. One of them is Straight Through Estimator (STE), which is proposed by Bengio et al. [21]. The function of STE is defined as follows:

$$clip\ (x\ , -1\ , +1) = \max(-1, \min(+1, x)) \qquad \textbf{(2.8)}$$

While x is the absolute value of the full-precision activation. The binary neural network can be directly trained through the STE using the same gradient descent method as the ordinary full-precision neural network. However, when the clip function is used in backward propagation, if the absolute values of full-precision activations are greater than 1, it can not be updated in backward propagation. Therefore, in the practical scenarios,

the identity function is also chosen to approximate the derivative of the sign function [20] [21].

However, an obvious gradient mismatch exists between the gradient of the binarization function (e.g., sign) and STE (e.g., clip). Moreover, binary neural networks suffer from the fact that the parameters outside the range of [-1, +1] will not be updated. To solve this issue, Yang et al. proposed Bi-Real, [18], a custom function (called ApproxSign) to replace the sign for back-propagation gradient calculation, it is defined as follows:

$$ApproxSign(x) = \begin{cases} -1, & if\ x < -1 \\ 2x + x^2, & if\ -1 \leq x < 0 \\ 2x - x^2, & if\ 0 \leq x < 1 \\ 1, & otherwise \end{cases} \quad \textbf{(2.9)}$$

$$\frac{\delta ApproxSign(x)}{\delta x} = \begin{cases} 2x + 2, & if\ -1 \leq x < 0 \\ 2 - 2x, & if\ 0 \leq x < 1 \\ 0, & otherwise \end{cases} \quad \textbf{(2.10)}$$

Compared to the traditional STE, ApproxSign has a compact shape (as shown in Figure 2.11) to that of the original binarization function sign. Therefore, the gradient error can be controlled to some extent**.** Circulant Binary Convolutional Networks (CBCN) [23] also applied an approximate function to address the gradient mismatch from sign function. Figure 2.11a shows the sign function and its derivative. Figure 2.11b illustrates the clip function and its derivative for approximating the derivative of the sign function proposed by Bengio et al. [21]. Figure 2.11c shows the differentiable piecewise polynomial function [18] [22] and its triangle-shaped derivative for approximating the derivative of the sign function in gradient computation.

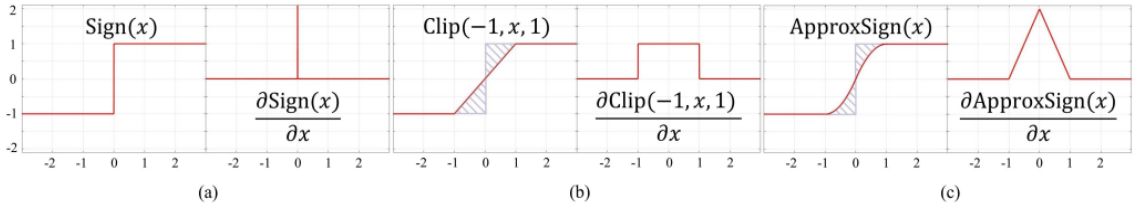


Figure 2.11 Different Functions Behavior in Backward Propagation [22]

Experiments on ImageNet show that the Bi-Real net, with the proposed training algorithm, can achieve up to 57% and 62% top-1 accuracy with 18 and 34 layers, respectively. Also, these experiments provide a comparison between memory usage of Bi-real net and full-precision Res-Net, shown in Table 2.4. BOP refers to the number of bit-operations in a neural network, where the calculation method is the same as calculating FLOPs for the floating-point operations in [69], excepting the operation calculated in BOPs is bitwise.

Table 2.4 Memory Usage and FLOPs Calculation in Bi-Real net.

		Memory usage	Memory saving	BOPs	Speed up
18-layer	Bi-Real net	4.2 MB	11.14×	1.04×10^{10}	11.06×
18-layer	Full-precision Res-Net	47 MB	-	1.16×10^{11}	-
34-layer	Bi-Real net	5.5 MB	15.97×	124×10^{10}	18.99×
34-layer	Full-precision Res-Net	87 MB	-	234×10^{11}	-

BinaryConnect [19] constrains the weight to either +1 or -1 during propagations. As a result, many multiply-accumulate operations can be replaced by simple additions and subtractions. As fixed-point adders are much more efficient in terms of area and energy than fixed-point multiply-accumulators, this is a huge gain. As mentioned before, binarization algorithms are aimed at transforming the real-value weights into the two possible values. The most common approach to performing this transformation is based on the sign function:

$$w_b = \begin{cases} +1, & if\ w \geq 0 \\ -1, & otherwise \end{cases} \quad \textbf{(2.11)}$$

Where w_b is the binarized weight and w is the real value. Although using the sign function helps us achieve all the benefits of binarization, it may lose a significant amount of important information during the training and potentially after training has been completed. For example, consider a situation where almost all network weights are in the range (0, p] where p is a positive number. In this case, all the weights will be transformed into +1. The same can happen for weights in the range (-p, 0). To avoid this situation, Bengio et al. [19] proposed an alternative strategy which is called stochastic binarization. Despite deterministic binarization, this approach transforms weights into binary values stochastically (Equation 2.12):

$$w_b = \begin{cases} +1, & with\ probability\ p \\ -1, & with\ probability\ 1-p \end{cases} \quad \textbf{(2.12)}$$

Depending on the network structure, training algorithm, quantization category, and training data, different approaches can occur. BinaryConnect can only support deterministic binarization and stochastic binarization to binarize the weight during the forward and backward propagation but not during the parameter update [24]. Bengio et al. [19] showed that stochastic-based binarization could obtain an almost 2% reduction in test error rates compared to regular DNNs on the CIFAR-10 benchmark image classification dataset. The algorithm that has been used in BinaryConnection during the backward propagation is based on STE. BinaryConnect uses a clip function to cut off the update range of the full-precision weights to prevent the real-valued weights from growing too large without any impact on the binary weights [19] [20] [21] [24] [81]. Lin et al. [25] showed that by designing a corresponding hardware architecture for Binary Neural Network, we could achieve better energy efficiency than traditional full precision representation. The post-layout results show that the proposed design can improve energy efficiency over 2.0TOPS/W which is two times better compared to prior methods. TOPS/W refer to Tera Operations Per Second per Power.

Rastegari et al. [32] proposed two approximations of standard CNNs: Binary-Weight-Networks and XNOR-Networks. The difference between these two approximations is about quantized parameters. In Binary-Weight-Networks, the filters are approximated with binary values resulting in 32x memory saving. In XNOR-Networks, both the filters and the input to convolutional layers are binary. XNOR-Networks approximate convolutions using bitwise operations. They showed that XNOR-net results in 58x faster convolutional operations and 32x memory savings. Rastegari et al. [32] point out that training is improved by placing the pooling layer after the BinConv layer rather than after the activation layer. Figure 2.12 provides a visual comparison between a typical block in CNN and a block in XNOR-Net. In this figure, BNorme, BinActiv, and BinConv refer to Batch Normalization, Binary Activation layer, and Binary Convolution layer, respectively.

Figure 2.12 Block Concentration (XNOR-Net vs Typical CNN) [32]

The Accurate Binary Convolutional Neural Network (ABC-Net) proposed by Lin et al. [37] is an attempt at increasing the accuracy gap between BNNs and full precision networks. The binarization of the weights is aided by calculating the mean of the weights. ABC-Net applies five binary activations to address activation quantization issues mentioned by Cai et al. [35] which is the need for back-propagation through the non-differentiable operators. By using this method, ABC-Net has reduced the Top-1 and Top-5 accuracy degradation caused by binarization to around 5% on ImageNet compared to the full precision counterpart.

Galloway et al. [26] proposed another benefit of very low-precision neural networks which is improved robustness against some adversarial attacks. An adversarial attack is a method of making small modifications to the objects in such a way that the machine learning model begins to misclassify them. They showed that in the very low-precision

cases where weights and activations are both quantized to -1 and +1 can sharply reduce the impact of iterative adversarial attacks. Their experiment illustrates that non-scaled binary neural networks exhibit a similar effect to the original defensive distillation procedure that led to gradient masking. Gradient masking is a term introduced in practical black-box adversarial attacks against DL systems using adversarial examples. Most adversarial example construction techniques use the gradient of the model to make an attack. For instance, attackers look at a picture of an airplane, they test which direction in picture space makes the probability of the “cat” class increase, and then they give a little push (in other words, they perturb the input) in that direction. The new, modified image is mis-recognized as a cat [70].

In summary, Table 2.5 provides a comparison between the accuracy of different Binary Quantized Networks and that of Full Precision on ImageNet dataset and different topologies [33].

Table 2.5 Accuracy Comparison of BNNs

Methodology	Topology	Top-1 Accuracy (%)	Top-5 Accuracy (%)
Original BNN	AlexNet	41.8	67.1
Original BNN	GoogleNet [68]	47.1	69.1
XNOR-Net	AlexNet	44.2	69.2
XNOR-Net	ResNet18	51.2	73.2
ABC-Net	ResNet18	65.0	85.9
ABC-Net	ResNet34	68.4	88.2
ABC-Net	ResNet50	76.1	92.8
Full Precision	AlexNet	57.1	80.2
Full Precision	GoogleNet	71.3	90.0
Full Precision	ResNet18	69.3	89.2
Full Precision	ResNet34	73.3	91.3
Full Precision	ResNet50	76.1	92.8

Chapter 3:

Methodology

As discussed in Chapter 2, four different approaches have been suggested to improve memory usage and computation costs for CNNs on devices with limited computational resources and memory capacity. The network pruning methods treat a network as a combination of different layers and operations like convolution layer, pooling layer, activation layer, etc. In contrast, the other three methods (transformation and hardware optimization, and parameter quantization) are aimed at improving the convolution operation as it is the costliest computation in CNNs.

We propose a method consisting of a combination of matrix optimization (based on the MEC method [10], parameter quantization (using Binary Quantization and applying it to pre-trained models we can decrease both computation costs and memory utilization at the same time), and transformation and hardware implementation (using software development techniques and data parallelism we provide an efficient implementation) approaches to provide a general solution that reduces computation costs and memory usage in the convolution operation and CNN subsequently.

Although we consider NVIDIA Jetson Nano 2GB as the target device, we developed our method using Open Computing Language (OpenCL) [43] instead of Compute Unified Device Architecture (CUDA) [71] for implementing the main components of our algorithm. OpenCL is an open, royalty-free, standard for cross-platform, parallel programming of device accelerators found in supercomputers, cloud servers, personal computers, mobile devices, and embedded platforms. The main OpenCL's competitor is CUDA, which is parallel computing platform and programming model created by NVIDIA. CUDA is built only for NVIDIA-produced hardware. Therefore, the implementation presented can be used on a wide device with little to no change in the coding structure.

In summary, the following topics will be covered in this chapter:

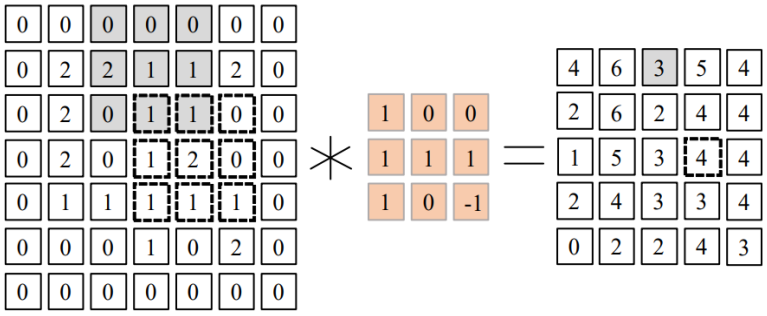
- Matrix Optimization
- Parameter Quantization
    - Quantization efficiency
    - Binarization efficiency
    - Quantization methodology
        - Training models method
        - Transformation function
        - Quantized parameters
- Software Implementation
    - Parallel computing (Task parallelism vs. Data parallelism)
    - OpenCL vs. CUDA
    - Implementation guideline
        - OpenCL programming model
            - OpenCL platform model
            - OpenCL memory model
        - Matrix representation
        - Host code and device code

3.1 Matrix Optimization

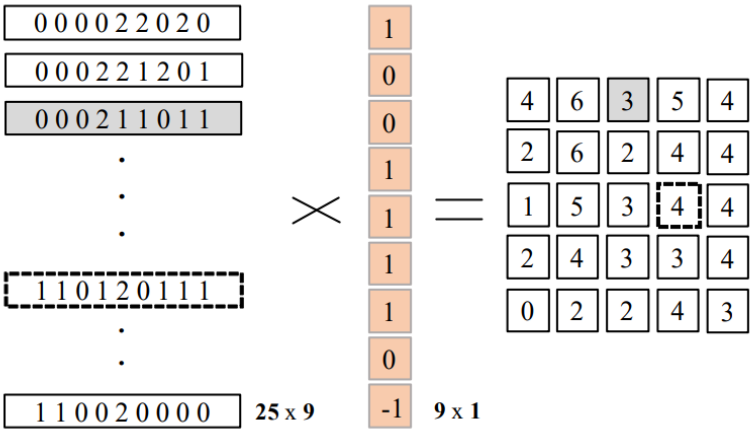
In Chapter 2, we presented an overview of seven different methods for performing a convolution operation and mentioned that there is always a trade-off between memory usage and computation time. For our implementation we were looking for an algorithm that brings noticeable performance improvements and is also compatible with a wide range of hardware.

As discussed in Chapter 2 the GEMM-based [52] algorithms are faster and more memory efficient than both the Fast Fourier Transform-based algorithms [13] [14] [12] and Winogard-based algorithms [15] both in terms of using either CPU or GPU processors. So, by using this fact, we scale down our possible choices into the GEMM-based algorithms set. To select the best algorithm, we studied GEMM-based algorithms (im2col [8], MEC [10]) in terms of memory overhead and execution time. We also compared the direct convolution calculation to have a scale for our conclusion. Figure 3.1 illustrates a

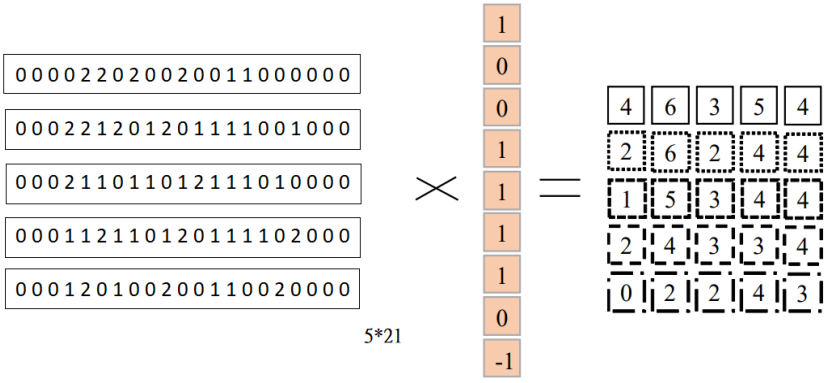
simple example of convolution operation for a 7 × 7 input matrix and 3 × 3 filter matrix, and the convolution stride is 1 × 1.



(a) Direct Convolution



(b) im2col Convolution



(c) MEC convolution

Figure 3.1 Memory Overhead Illustration [10]

The outputs of Figure 3.1 convolution operations are the same 5×5 matrix. Theoretically, as direct convolution does not use a lowered matrix, its memory overhead is zero for any input and filter size. For any input and filter size, the lowered matrix of im2col method is as follow:

$$im2col\ lowered\ matrix = (output\_width \times output\_hight \times input\_channel) \times (filter\_width \times filter\_hight \times number\_of\_filter) \quad \textbf{(3.1)}$$

Which is 25×9 in this example, this number is decreased to 5×21 in the MEC lowered matrix. In MEC, the number of lowered matrix elements is calculated by using Equation 3.2.

$$MEC\ lowered\ matrix = (output\_width) \times (filter\_width \times input\_hight \times input\_chanel \times number\_of\_filters). \quad \textbf{(3.2)}$$

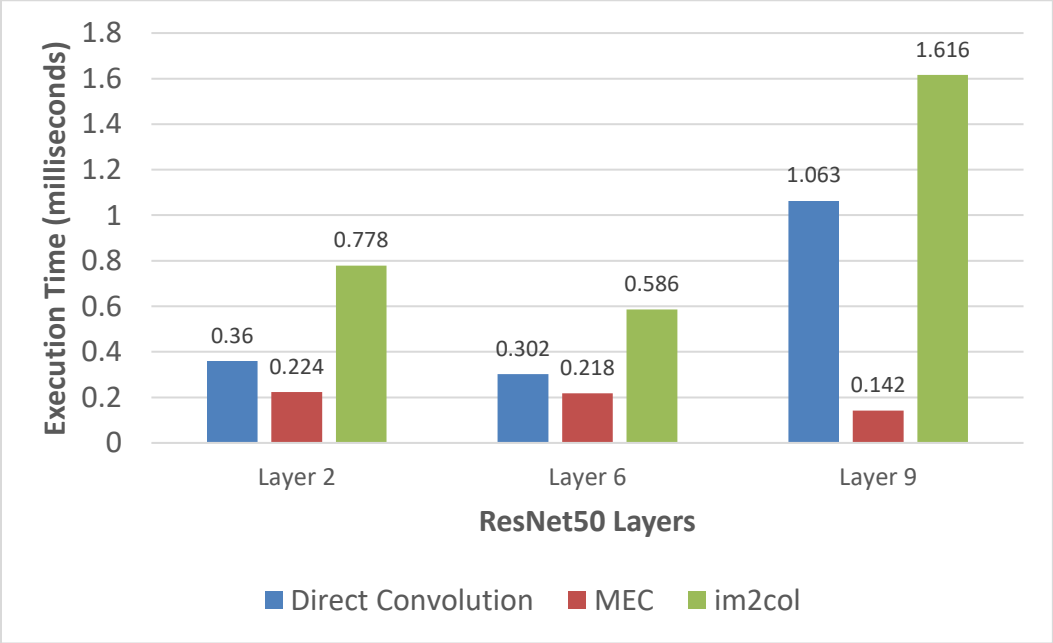
Note that in this specific example, matrices are two dimensional which means that *Number_of_filters = 1, input_channel = 1, number_of_filters = output_chanel = 1.*
To show the difference between these three algorithms (direct convolution, im2col, and MEC), we perform experiments that test algorithms under different scenarios (different inputs and filters). In each scenario, we consider a convolution layer of ResNet50 as our reference for input matrix size and filter size. Also, these matrices (input matrix and filter matrix) were created by random floating-point variables. This experiment was based on single thread implementation, which means that we only used one CPU core to execute implementation of each algorithm. Table 3.1 illustrates the inputs and filters sizes of each corresponding layer. Then we execute each algorithm with the mentioned input sizes. Figure 3.2 illustrates the experimental results for execution time and memory usage, respectively. From the results presented in the bar graphs, we can conclude that:

- MEC is faster than im2col and direct convolution.

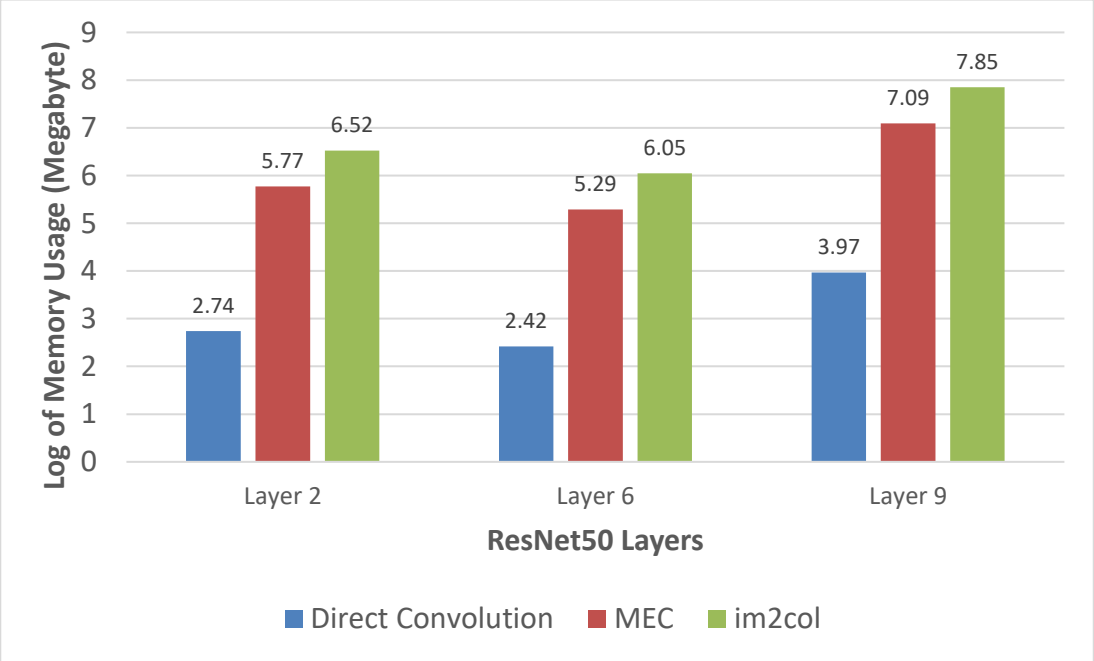
- MEC uses less memory than im2col.
- Im2col is not an efficient solution for the convolution operation if the executional environment is a single core processor.
- For single core processor which does not benefit the parallelism, either direct convolution or MEC can be chosen. In this case, if the restriction is the memory usage, we can pick direct convolution. But if the restriction is runtime, MEC could be better solution.
- As the size of filters increases and the stride gets bigger, MEC can carry out operations even faster.

Table 3.1 Convolution Operation Parameters of Three of ResNet50's Layers

Layer number	Input size	Filter size	Stride	Padding
Layer 2	224*224*3	3*3*3*64	2*2	3*3*3*3
Layer 6	256*256*1	3*3*1*64	1*1	0*0*0*0
Layer 9	127*127*64	3*3*64*64	1*1	1*1*1*1



(a) Execution Time Comparison Between Direct Convolution, im2col, and MEC



(b) Memory Usage Comparison Between Direct Convolution, im2col, and MEC

Figure 3.2 Performance of Direct Convolution, im2col, and MEC

As im2col and MEC are using a lowered matrix to perform the operation, both have memory overhead in comparison with direct convolution.

Although im2col is not an efficient solution even in comparison with direct convolution in this experiment, on multicore hardware with the benefit of parallelism, we obtain results for execution time. For showing this fact, we designed the same experiment implemented with parallelism. We noted that memory usage did not change in the second experiment. Figure 3.3 provides a visual comparison of algorithms runtime on NVIDIA GeForce GTX960 GPU. In this experiment all parameters are the same only the environment is changed, and the implementation is based on data parallelism.

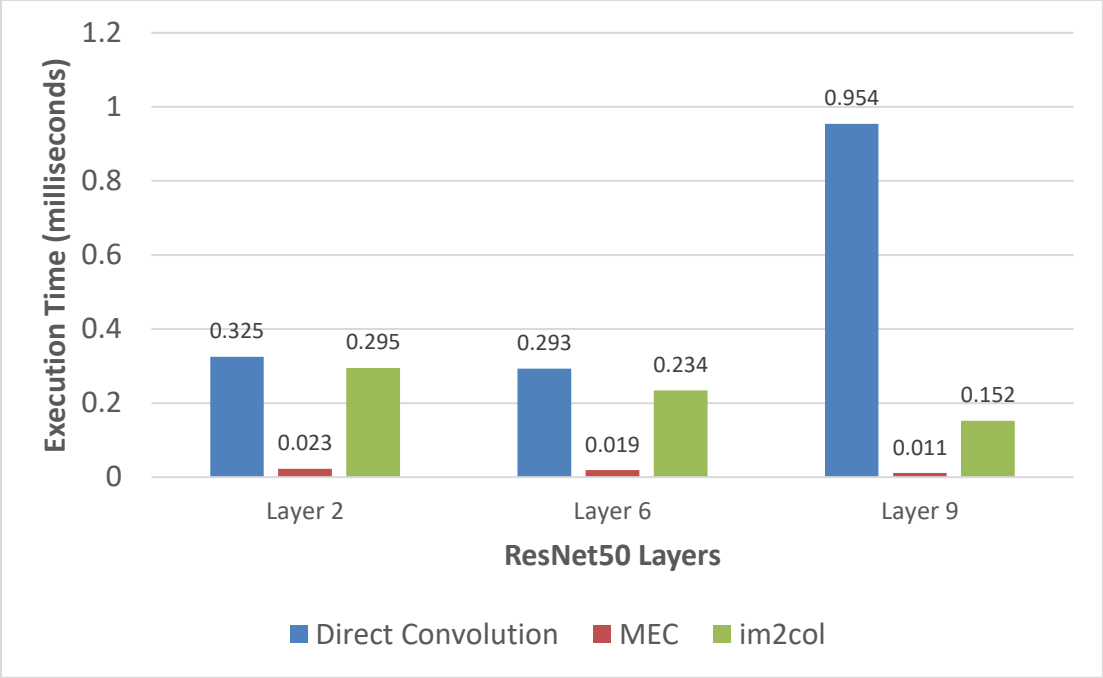


Figure 3.3 Execution Time Comparison Between Direct Convolution, im2col, and MEC

Based on the results of this experiment and the fact that we plan to use multicore hardware, we have selected the MEC algorithm as the main component of the convolution operation.

3.2 Parameter Quantization

The next step in our solution is to find an appropriate quantization method. In Chapter 2, we discussed how parameter quantization could be used to improve the runtime of CNNs and reduce memory usage at the same time. We also provided examples of previous work which implemented networks by applying this method. This section will cover why quantization is effective and why binary quantization can be useful.

3.2.1 Quantization Efficiency

DNNs have a considerable number of parameters which do not have the same level of importance. As mentioned by Denil et al. [38], in the best cases, more than 95% of the parameters in a neural network can be predicted without any drop in accuracy performance. In work from Molchanov et al. [39] their model compression showed that nearly 99% of weights could be pruned in most types of neural networks.

These research and many others [53] [54] [40] showed that quantization is potential answer to address how we can reduce memory usage and computation complexity of a CNNs without any significant hit on their accuracy.
Therefore, we choose Quantization to be a part of our general solution. As mentioned in Chapter 2, quantization methods can be categorized based on the quantized data type. In this project, we decided to use Binary Quantization and we outline our reasons in the following chapters.

3.2.2 Binarization Efficiency

Between fixed-point quantization, integer quantization and binary quantization, we choose binary quantization over the others. The main reasons for making this decision are listed below:

1. Although fixed-point/integer quantization methods provide faster multiply computation and less memory usage than floating-point precision, we need to be concerned about the computation and memory costs of performing this type of quantization. Theoretically, fixed-point/ integer operations are more efficient than full precision, but it has extra cost and overhead to transform floating-point parameters into fixed-point/integer data types.
2. By using binarized parameters, we can use bitwise operations like XNOR and bitcount to perform a convolution operation instead of regular operations, which are by far slower than bitwise operations.
3. As we can use binary operation because of binary quantization, the final method would be more hardware friendly. It can be implemented in a large range of hardware such as FPGAs, Intel processors, and NVIDIA Developer Tool Kits, etc.
4. It is evident that 1-bit quantization require less memory than n-bit quantization methods (where n is any number greater than 1). This means binarization improves memory usage of CNNs and computation speed by using bitwise operations and helps improve runtime by providing faster data flow as shown by Ghimire et al [84].

5. Essert et al. [41] showed that a binary activation can use sharply increasing response for event-based computation and communication (consuming energy only when necessary) and therefore is energy-efficient. This cannot be employed in the fixed-point/integer-based schemes.

3.2.3 Quantization Methodology

We have discussed why we choose binary quantization over other types of quantization. In this section, we are going to look at the details of our binarization algorithm. Assume that we have *I* as input and *F* as the filter to the n-th convolution layer of a CNN. By applying the convolution operation *(I conv F)*, we reach the output of the current layer which is *O*. What is apparent is that the output of the n-th layer is the input of the n+1-th layer of the network and the input of the n-th layer is the output of the n-1-th layer. In the CNNs training process, for the forward pass, we move across the CNN, moving through its layers, and at the end obtain the loss, using the loss function. By using the backward pass, we propagate the loss from one layer to another. Figure 3.4 illustrates a general scheme of data flow during a training process for a convolutional layer. In this illustration, *I*, *F,* and *O* are referring to input, filter, and output, respectively. In this figure, the red arrows represent the flow of backward propagation, and the black arrows show data flow in forward propagation.

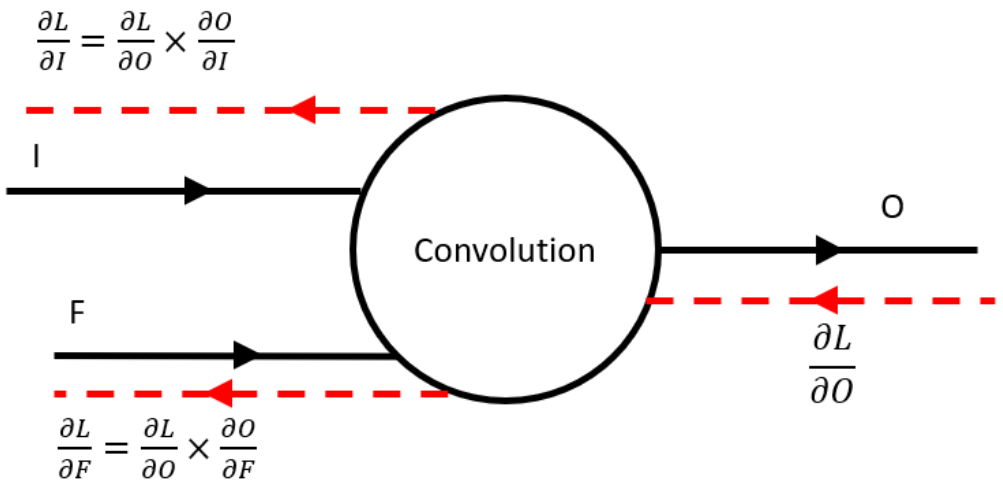


Figure 3.4 Dataflow of the Training Process in Convolution Layer

During backpropagation, we obtain the gradient of the loss from the previous layer as $\partial L/\partial O$**.** For propagating this loss to the previous layer, we need to calculate $\partial L/\partial I$**.** As we probably have no idea about the relationship between I and L, we use the chain rule to calculate $\partial L/\partial I$**.** Likewise, we need to calculate $\partial L/\partial F$ to update the filters. Filters are the learnable parameters in the CNNs which means they need to be updated in each training cycle. Equation 3.1 shows how the filters are updated. In this equation, α is the learning rate. The learning rate is a hyperparameter that controls how much to change the model in response to the estimated error each time the model weights are updated.

$$F_{updated} = F - \alpha(\partial L/\partial F) \quad \textbf{(3.1)}$$

Forward and backward propagation is a mutual process in every learning strategy regardless of the type of network architecture or the type of training data. When we want to apply binary quantization to a network, we face two big issues:

- **Differentiability**

As mentioned in Chapter 2, the biggest issue for dealing with quantization methods is providing a differentiable function as an activation function. As we use 1-bit quantization, there are two possible values for each parameter. This means that the activation function results in an impulse function which is not useful during learning process.

- **Filter Update**

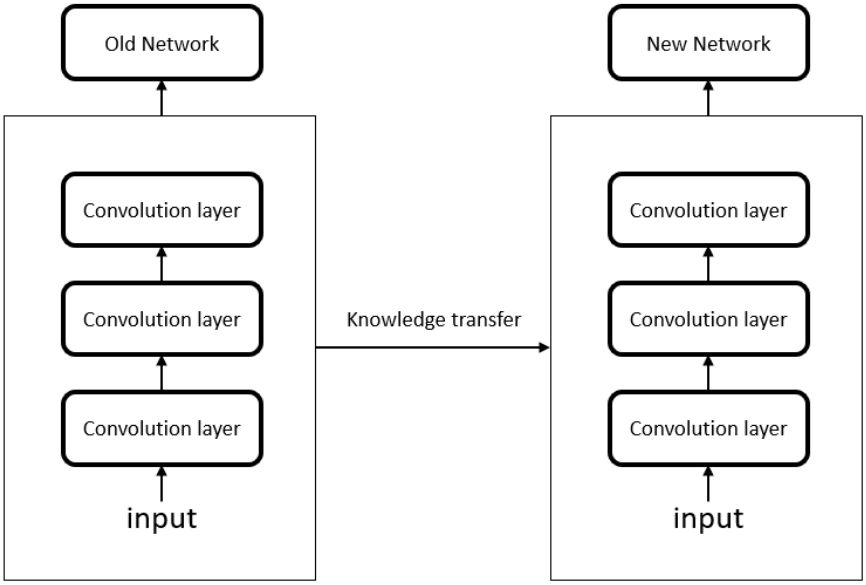
Value of $-\alpha(\partial L/\partial F)$ is mostly a small value, which means that during a training process, filters are changed slightly by adding $-\alpha(\partial L/\partial F)$. The issue is that F can accept only two values. Making changes to F means toggling between these two values. So, a small value of $-\alpha(\partial L/\partial F)$ does not have enough power to update filters' values.

These mentioned issues are the most common reason and motivation for researchers to provide new strategies to train BNNs. As discussed in the Chapter 2 in detail, researchers have proposed different activation functions and learning algorithms to hit the issues. Our contribution to addressing the mentioned issue is classified into three groups.

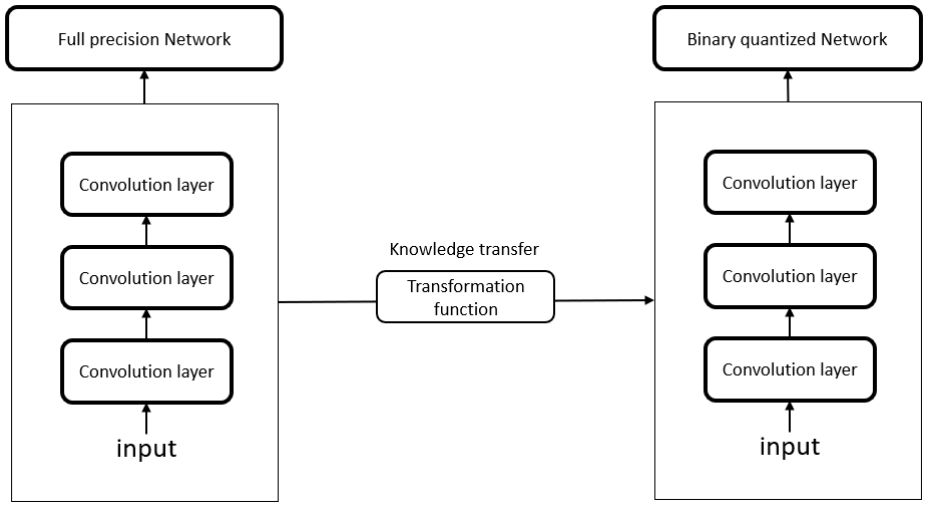
1. Step 1 – A strategy for creating and training a CNN.
2. Step 2 – A transformation function for reforming a CNN to BNN.
3. Step 3 – Determine which parameters can be binarized using our method.

3.2.3.1 Training Models

To avoid all the complexities related to training a BNN, we treat the network as a full precision network during the training process. This means that we train networks with floating-point values. Although training large networks like Mask-RCNN, SSD, etc. is a time-consuming task, it is not done frequently and therefore our training model is a branch of transfer learning. Transfer learning refers to the reuse of a pre-trained model on a new problem. A machine uses the knowledge learned from a prior assignment to increase prediction about a new task. The main goal of transfer learning is to reduce training time, improve neural network performance in terms of accuracy, and reduce the effect of the lack of a rich training dataset. In this case, our concern is not about training time and the absence of a large amount of training data, it is about accuracy. Therefore, we will train a full precision network by using a training set and during the transforming data convert them to binary format. Figure 3.5 illustrates the main difference between our learning model and traditional transfer learning.



(a) Schema of Transfer Learning



(b) Schema of Our Proposed Learning Method for Binary Neural Networks

Figure 3.5 Illustration of Learning Models

By using this strategy, we do not need to deal with the training issues that come with only using binary operations for that training. The only remaining concern is the transformation function. To address this issue, we propose a method to convert a floating-point trained network to a binary network. This transfer method is called *shiftSign* and is discussed in the next section**.**

3.2.3.2 Transformation Function

As with other parts of our method, we use -1 and +1 to represent the binary values. The main reason behind choosing binarization is that we want to replace multiplication and addition operations with XNOR, and bit count respectively. According to the truth table (Table 3.2), we can easily replace multiplication with XNOR. For showing that we can replace multiplication operations with the XNOR we just need to consider -1 and 1 as real values of encoded 0 and 1 values. In other words, the binary values can either be -1 or +1, and these signed binary values are encoded with a 0 for -1 and a 1 for +1. To be clear, we name the values -1 and +1 as binary “values” and their encoding, which are 0 and 1, as binary “encoding”.

Table 3.2 Truth Table of XNOR Operation

Encoding (Value)	Encoding (Value)	XNOR(Multiply)
0(-1)	0(-1)	1(+1)
0(-1)	1(+1)	0(-1)
1(+1)	0(-1)	0(-1)
1(+1)	1(+1)	1(+1)

So far, we have discussed why binary "value" and binary "encoding" are essential to understand the concept. Most of the proposed methods for creating a binary neural network are using *Sign* function as the transformation function. Equation 3.2 shows how the Sign function works. There is no guarantee about the distribution of weights which means that maybe there are some pre-trained networks in which all trained parameters are greater than zero or less than zero. Generally speaking, the distribution of parameters is important, even though after studying them, we reach the same transformation function as Sign.

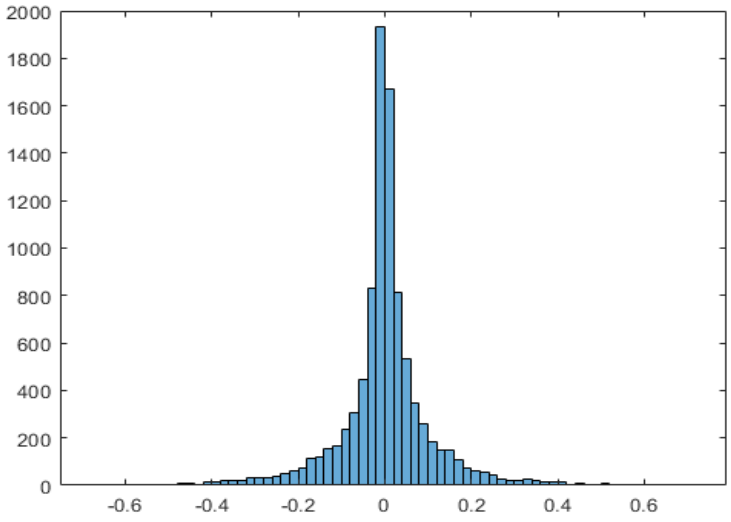
$$Sign(x) = \begin{cases} -1, & x < 0 \\ +1, & x \geq 0 \end{cases} \quad \textbf{(3.2)}$$

Our proposed transformation function which is named *shiftSign* is defined as the following:

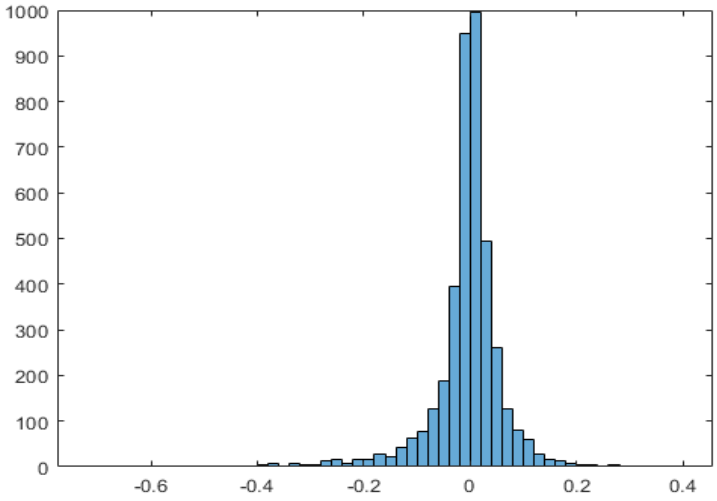
$$shiftSign(x) = \begin{cases} 0\,, & x < \beta \\ 1\,, & x \geq \beta \end{cases} \quad \textbf{(3.3)}$$

The parameter β is a hyperparameter that can be found by studying learned filters in the real value pre-trained network. A combination of these two mentioned strategies (the proposed learning method and *shiftSign*) is highly beneficial to our proposed binarization algorithm in terms of maintaining accuracy at an acceptable level and reducing the rate of

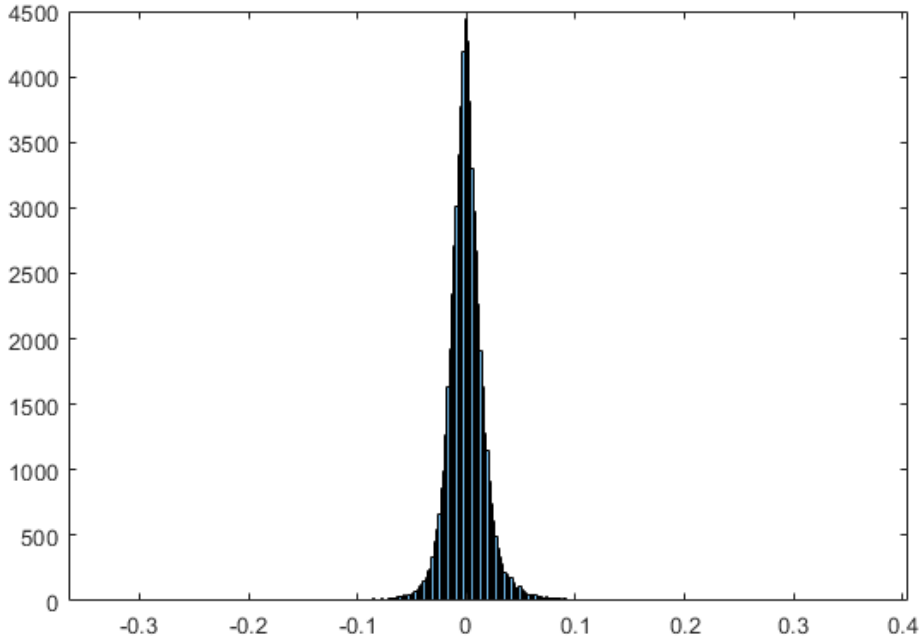
loss of knowledge. The point is no matter which parameters (filters or inputs) are transformed by *shiftSign*, the value of β is computed by averaging full precision filters of each convolution layer. By studying the distribution of parameters, we can choose a point that can be the toggle point. Figure 3.6 illustrates the distribution histogram of weights of pre-trained ResNet50. In this example, the network was trained by the ImageNet dataset. According to the histograms, the best value for β is zero.



(a) The 2nd Layer



(b) 6th Layer



(c) 60th Layer

Figure 3.6 Illustration Weights' Distribution.

So far, we mentioned our proposed method in terms of the binarization process. In the next section, we discuss our approach to select quantizable parameters.

3.2.3.3 Quantized Parameters

We can use the transformation function in each convolution layer for either the input matrix or the filters or both. If we use the transformation function just for either input or filters, we will not benefit from all bitwise operation advantages, which means that although we transformed one of the matrices into 1-bit representation, we cannot apply XNOR and bit count operations. In these cases, we would simply decrease memory usage and make slight improvements in computation speed. In the most well-trained networks, it is possible to transform both input and filter into the 1-bit representation, as we discussed before. In this project, we use Mask-RCNN and MNIST for evaluating our proposed algorithm. We apply the transformation function on both inputs and filters. In these two networks, we obtain good results in terms of accuracy, memory usage, and runtime.

3.3 Software Implementation

Although the proposed algorithm is fast and memory-efficient, a wrong implementation can deprive us of its benefits. In addition, none of the machine learning frameworks such as PyTorch or TensorFlow [86] support binary parameterization. Both facts led us to provide an implementation for our method. Obviously, it was not the first time that someone provided a parallel-processing-based implementation for convolution operation, but it is the first time that the mentioned operation is binarized and used MEC at the same time. What makes this implementation different, from previous parallelized implementations, is its combination with binary quantization, MEC, wide range of hardware considerations, and its efficiency in terms of data transformation between different processors.

Our contribution in this part is to provide a parallel-processing-based implementation that can be run on a large set of hardware regardless of the manufacturer.

In this section, we will discuss the fundamentals of parallel computing. After that, we will provide a comparison between different tools which are available for parallel-processing-based application programming. Lastly, we will go straight into implementation-related points like matrix representation, heterogeneous programming, etc.

3.3.1 Parallel Computing (Task parallelism vs Data parallelism)

Parallel computing is referred to a type of computation in which many calculations or processes are carried out simultaneously. Parallel computing is slightly different from multithreaded programming. In parallel computing, the target device must have multiple cores in the processor. In fact, parallel computing is an effort to use all the cores to carry out tasks at the same time and parallel computing on a single-core processor is meaningless. Whereas by using scheduling algorithms, multithreading can be carried out on a single processor, and it gives the illusion of running in parallel. Parallel computing can be categorized into different groups based on implementation point of view: hardware implementation and software implementation. From the hardware implementation point of view, it is categorized into 4 groups which are bit-level

parallelism, Instruction-level parallelism, Task parallelism, and Superword level parallelism [72]. In this project, we look at parallel computing from a different point of view which is under software implementation. From a software implementation point of view, parallel computing is divided into two parts: Task parallelism and Data parallelism. In the rest of this section, we are going to cover the differences between Task parallelism and Data parallelism. Furthermore, we will discuss why we choose Data parallelism over Task Parallelism for the purposes of this project. Task Parallelism means concurrent execution of the different tasks on multiple computing processors, whereas Data Parallelism refers to concurrent execution of the same task on each multiple computing units. Task parallelism focuses on executing different operations in parallel to fully utilize the available computing resources in form of processors and memory. One common example of task parallelism would be the situation in which the application creates threads for doing parallel processing where each thread is responsible for performing a different operation. The following pseudo-code illustrates task parallelism:

```
For each CPU in a parallel computing environment
{
       Task task = Retrieve next task from task queue;
       Thread thread = Create a new thread;
       thread.assign(task);
       thread.start();
}
```

On the other hand, in Data parallelism, the same operation is performed on different parallel computing processors on the distributed data subset. The following pseudo-code illustrates data parallelism:

```
lower_limit = 0;
upper_limit = 0;
For each CPU in parallel computing environment
{
       lower_limit = upper_limit + 1 ;
       upper_limit  = upper_limit + round (data_array.length / no_of_cpus);
       Data data = data_array[lower_limit :  upper_limit];
       Thread thread = Create a new thread;
```

```
        thread.feed(data);
        thread.start();
}
```

Table 3.3 shows the main difference between task parallelism and data parallelism. The point of using MEC is converting the convolution operation into matrix multiplication form, because by using the GEMM concept, multiplication can be performed in parallel. In fact, matrix multiplication is an iterative process in which its task is the same and just data will be changed. Therefore, we use the Data parallelism concept in the provided implementation.

Table 3.3 Difference between Data Parallelism and Task Parallelism.

Data Parallelisms	**Task Parallelisms**
Same tasks are performed on different subsets of the same data.	Different tasks are performed on the same or different data.
Synchronous computation is performed.	Asynchronous computation is performed.
As there is only one execution thread operating on all sets of data, so the speedup is more.	As each processor will execute a different thread or process on the same or different set of data, so speedup is not as significant.
The amount of parallelization is proportional to the input size	Amount of parallelization is proportional to the number of independent tasks performed
It is designed for optimum load balance on a multiprocessor system.	Load balancing depends upon the availability of the hardware and scheduling algorithms like static and dynamic scheduling

3.3.2 OpenCL vs CUDA

We discussed that between task parallelism and data parallelism, data parallelism can meet our goal for using GEMM. Also, we want to use all available processing cores as

efficiently as possible. To do this, we need to apportion the processing load between all processing cores. Nowadays, even low-power hardware has multi compute units, which is the ideal environment for parallel computing. For instance, the NVIDIA Jetson NANO 2GB has 128 processing cores in its GPU. To benefit from this processing power, we need to use APIs that provide this capability for us. Two available tools for parallel computing are CUDA and OpenCL. Unlike CUDA which only runs on NVIDIA devices, OpenCL is not hardware-based and can run on variety of hardware, and this makes it a good choice for heterogeneous computing. Heterogeneous computing is a term that refers to systems that use more than one kind of processor or core. Among CUDA and OpenCL we choose to implement our proposed method based on OpenCL. The main reason for making the decision is that using CUDA restricts the implementation to only running on NVIDIA hardware. Furthermore, as OpenCL is open, companies have developed it to make it more powerful in their hardware. For example, Xilinx has introduced a vendor extension on top of the OpenCL 2.0 specification for loop pipelining [44].

3.3.3 Implementation Guideline

Our implementation is based on pure C programming language without any use of C++ or other languages. This part is not about how to code in C or any references to implementation details. In this section, we are going to introduce some important points related to the OpenCL implementation of the method. First, we focus on the OpenCL programming model background which is important for understanding what is going on during parallel processing. After that, we discuss different ways to define a matrix. Lastly in the two last parts, we are trying to paint a word picture of our implementation.

3.3.3.1 OpenCL Programming Model

This section contains fundamentals about the OpenCL programming model. In this part, we are going to clarify different memory levels in an OpenCL application, also we discuss how data flow would be in this kind of application. It is necessary to mention that we are trying to avoid addressing programming notes or syntax-related issues and keeping everything as abstract as possible.

An OpenCL application is split into host and device parts. The host part code is written using a general programming language such as C, C++, Java, etc. As mentioned, we used C and the code was compiled using a conventional compiler for execution on a host CPU. The host part is an interface that provides all needed communication between the device (parallelized part) and the rest of the application. In other words, the host code is the part that is used to transfer data between the host hardware and the OpenCL device. In the host code, developers can specify the type of OpenCL device, which can be CPU, GPU, Accelerator, all available devices, and the default OpenCL device in the system. For example, if the OpenCL device is an FPGA (like Xilinx, Intel, etc.), the developer should choose Accelerator as the OpenCL device in the host application. Which means that Accelerator is a generic name for FPGA type implementations.

OpenCL code is written in a text file which is mostly formatted as a ".cl" file. Each defined function in the ".cl" file is known as a "kernel". The ".cl" file is loaded in the host code and depending on the chosen OpenCL device it would be compiled at runtime. Figure 3.7 illustrates the difference between traditional and OpenCL programming, it also shows the connection between the host part and the device part. In this illustration myapplication.c is the host code.

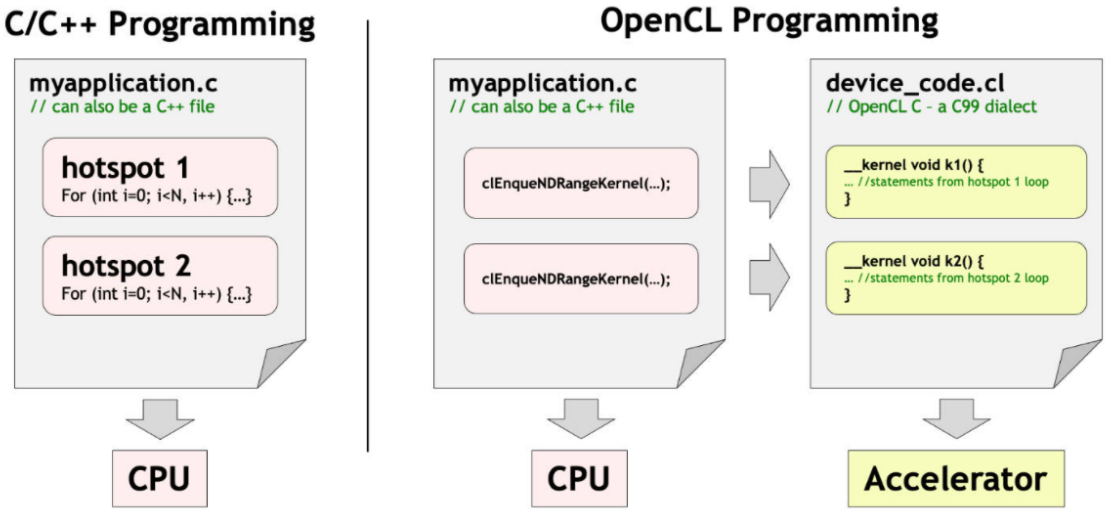


Figure 3.7 Traditional vs OpenCL Programming Paradigm [43].

OpenCL Platform Model

From OpenCL hardware point of view, the OpenCL platform is a model shown in Figure 3.8 which consists of a host processor connected to one or more compute devices (OpenCL devices). An accelerator or a set of CPU cores is configured as a compute device. Each compute device contains one or more compute units (CUs) and has the compute device memory. The device memory of a compute device is not visible to other compute devices and consists of the global memory and the constant memory. Constant memory is read-only for the compute device. Each CU contains one or more processing elements (PEs) and the local memory. A PE is a processor and has its own private memory. PEs in the same CU share the local memory, and the local memory is not visible to other CUs. From the OpenCL software point of view, the smallest processing unit of an OpenCL device is a work-item. Each work-item in OpenCL is a thread in terms of its control flow and its memory model. Work-items are grouped into workgroups. A work-item executes on a PE, and each workgroup executes on a CU.

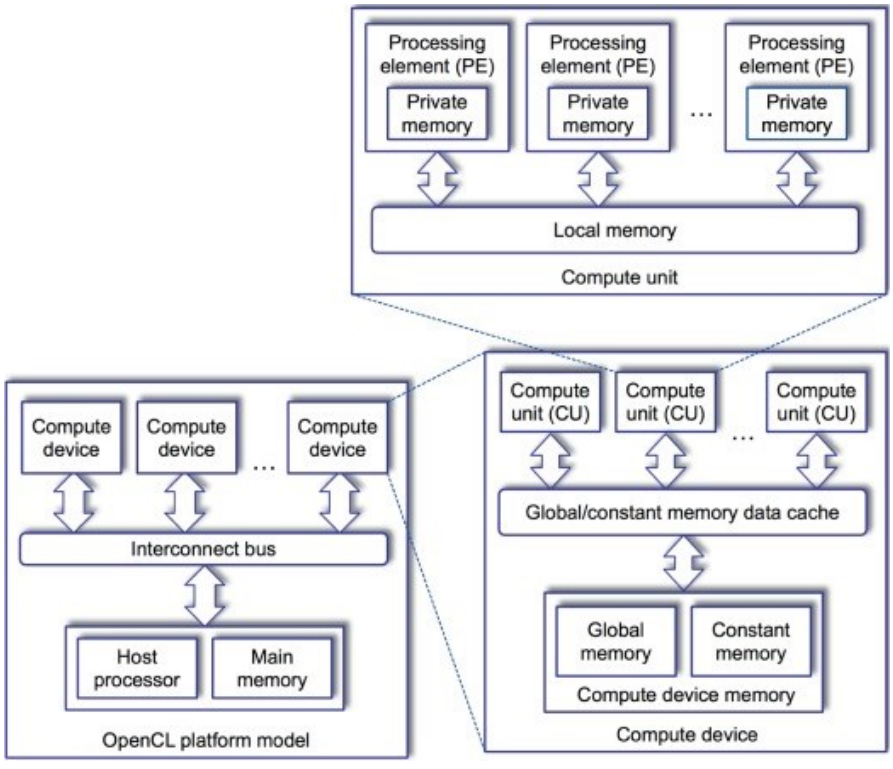


Figure 3.8 Generalized OpenCL Device Structure [57]

OpenCL Memory Model

An OpenCL device has four memory domains which are: private, local, global, and constant. Figure 3.9 provides a visual illustration of these different domains. Figure 3.9.a shows how the OpenCL memory model maps to specific hardware units in an OpenCL device and Figure 3.9.b shows data flow in an OpenCL device. In this illustration, LDS refers to Local Data Share.

1. **Global Memory**

The global memory lets the host application read from and write to. Global memory is the main way that the host transfers data to the OpenCL device (Figure 3.9.b). As shown in Figure 3.8, global memory is shared with all compute units (workgroups) and processing elements (work-items).

2. **Constant Memory**

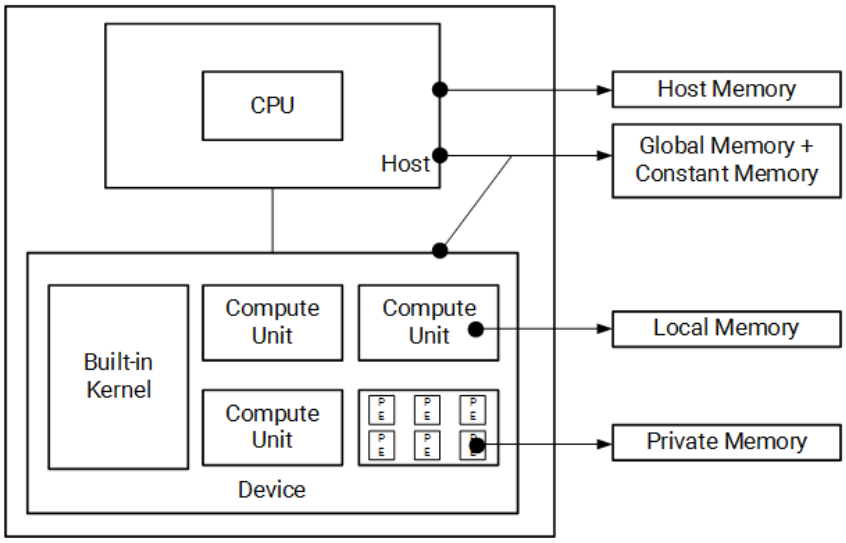
Constant memory is defined as the region of the OpenCL device that is accessible with read and write access for the host and read-only for CU. As the name implies, Constant Memory is typically used to transfer constant data, which are required for kernel computation, from the host to the OpenCL device.

3. **Local Memory**

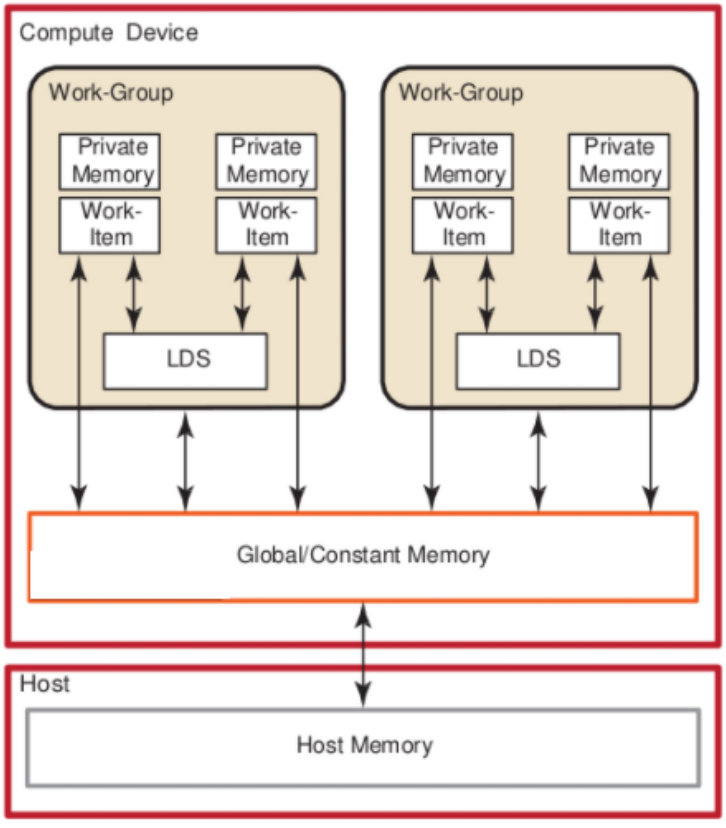
Local memory is the region of OpenCL device memory that is local to a single CU (workgroup). The host has no access and no visibility to local memory. This memory allows read and write operations by PEs (work-items). This memory space is typically used to store data that must be shared with other work-items (Figure 3.7 and Figure 3.9.b).

4. **Private Memory**

Private memory is defined as the region of memory that is assigned to an individual work-item within an OpenCL processing element. The host has no access and visibility into this memory region. Also, this memory space can be read from and written to by a work-item. Variables defined in one work-item's private memory are not visible to and accessible to another work-item.



(a)



(b)

Figure 3.9 OpenCL Memory Model [77]

3.3.3.2 Matrix Representation

Different representations have been suggested to define a matrix. A software developer can use different data structures for defining a matrix. The most common data structure for define a matrix is array. In this research, we consider array representation of a matrix. In most programming languages, matrix is known as multi-dimensional arrays. This representation is the most readable and developer-friendly representation. The other representations consider a matrix as a one-dimension array and by using other parameters like width and height to determine its size. For example, consider a three-dimensional matrix. Table 3.4 shows three different possible representations of it.

Table 3.4 Illustration of Three Different Ways for Representing a Matrix.

<table>
<tr><td colspan="3">The first channel:
<table><tr><td>1</td><td>2</td><td>3</td></tr><tr><td>4</td><td>5</td><td>6</td></tr></table>
The second channel:
<table><tr><td>7</td><td>8</td><td>9</td></tr><tr><td>0</td><td>1</td><td>2</td></tr></table></td></tr>
<tr><td colspan="3">Visual matrix</td></tr>
<tr><td>[[[1,2,3], [4,5,6]],
[[7,8,9], [0,1,2]]]</td><td>[1,2,3,4,5,6,7,8,9,0,1,2,]
W = 3, h=2, c=2</td><td>[1,4,2,5,3,6,7,0,8,1,9,2]
h=2, w=3, c=2</td></tr>
<tr><td>Multidimensional array</td><td>Rows first</td><td>Columns first</td></tr>
</table>

Our experiments show that although none of these representations has huge effect on MEC execution time, a “rows first” representation is slightly faster than others and that is because of the way that memory is addressed and the way we create lowered matrix. Also, as shown in Figure 2.6, each element of output matrix is generated by multiplying

part of the lowered matrix's rows elements and the filter. By using a "rows first" representation, it is less complicated to access the rows in terms of programming.

3.3.3.3 Host Code and Device Code

Same as every OpenCL-based application, this work is split into the host and the device parts. We define three kernel functions inside the OpenCL file. One of them is defined for extending the input matrix with zero padding if needed, another one is declared for creating the MEC lowered matrix, and the last one is used for performing the MEC multiplication operation. In the host part, we only have two public methods, the first one is the "init()" function, which is used to load the OpenCL code file and compile it, and the second one is "conv()", which receives two matrices as input and other convolution operation parameters like stride and padding. Figure 3.10 illustrates the host part and data communication between host and device.

In this figure, the blue arrows illustrate data flow, and the red boxes represent the kernel code. The key point here is avoiding unnecessary data communication. Copying data from the host memory to the global memory is time-consuming. By optimizing the transition process, we attempt to decrease the latency. Our strategy is to perform every process in the OpenCL device and only load the result after processing has completed.

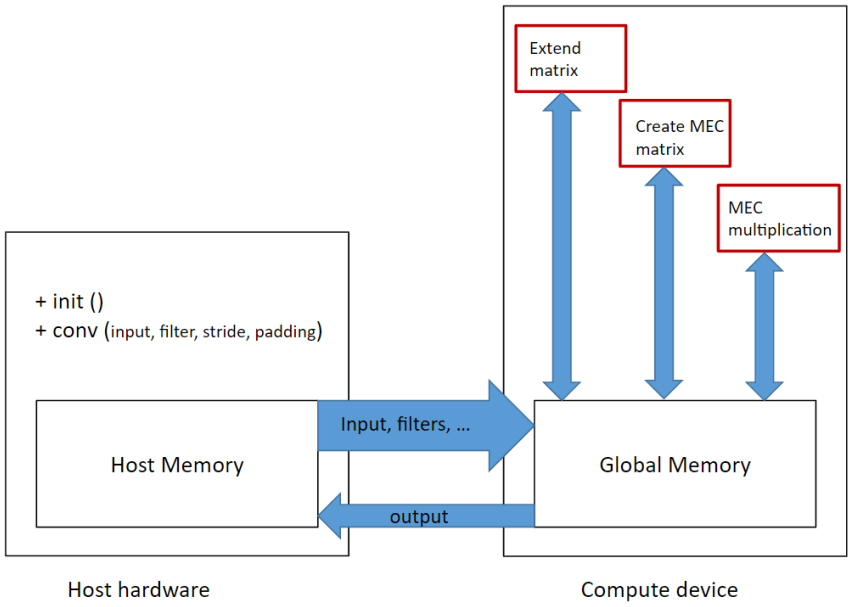


Figure 3.10 Illustration of Data Transfer in the Implementation

3.4 Conclusion

In conclusion, our proposed method is an aggregation of three different approaches for addressing CNN optimization concerns. From the matrix optimization point of view, we carried out many experiments to choose between different available algorithms for operating the convolution operation. The experiment's results showed that MEC is the best solution among all those presented (im2col, FFT, etc.). From the quantization point of view, our research showed that binary quantization could satisfy the problem criteria. We defined a new strategy for converting well-trained CNNs from full precision parameters into a quantized network. We defined our transfer function *shiftSign***.** And lastly, by using the OpenCL programming Technique, we tried to take advantage of parallelism. What makes this work different from previous work is the approach to the CNNs' performance issue, previous work considered only one approach to address the issue, while in this work, the problem is addressed by a composition of different approaches. Also, for the first time, we combined MEC and binarization for the first time. The outcome of this contribution is an efficient implementation of binary convolution operation and a methodology for converting a full precision Neural Network to a Binary Neural Network.

Chapter 4:

Results

The main motivation of this thesis has been defined as an effort to decrease resource utilization of CNNs. In Chapter 2, we presented different strategies that may satisfy this objective. As discussed in Chapter 3 we proposed a method to optimise the convolution operation as it is the most resource consuming part of any CNN. Our method contains three different stages: (1) an optimal way to perform the convolution operation of two matrices. (2) binary quantization to decrease runtime speed and memory usage, and (3) a parallel implementation for the method. To evaluate the proposed method, we need to show that it could be effective in terms of accuracy and resource usage in various scenarios. First, we consider the method as a convolution operation and show its performance as a single operation. Second, the method was considered as a convolution layer in various networks. We considered these two scenarios to test our method as a operation and test it while it has connection with other layers and functions in a real environment. This chapter contains the results of the different tests and experiences on the method performance.

In summary, the topics will be covered are:

- The Method as a Convolution Operation
    - Timing Analysis
    - Memory Analysis
- The Method as a Convolution Layer
    - Simple Network Accuracy
    - Complex Network Accuracy

4.1 The Convolution Operation

In this section, we will discuss the tests executed on the proposed method as a convolution operation in terms of timing and memory usage. To provide a clear view of

the method's performance, we also tested all the test cases under the same conditions on PyTorch. PyTorch is an open-source machine learning framework that is a python-based scientific computing package that uses both the power of GPUs and CPUs [75]. Among all developed machine learning frameworks, PyTorch is considered the most efficient and used by some of well-known deep learning libraries such as Detectron2. Choosing PyTorch as a standard implementation for convolutional operation helps us to provide a fare comparison in terms of runtime and memory usage. To provide the same condition during the test, we used the same development environment target device, the NVIDIA Jetson Nano 2GB. Topics in this section are categorized into two groups: timing analysis and memory analysis.

4.1.1 Timing Analysis

In the Timing analysis, we tested both the PyTorch convolution operation and our OpenCL-based implementation of the proposed method with various inputs. In this experiment, we chose matrices as inputs whose sizes are most frequently used in key networks such as Mask-RCNN [56], ResNet101 [63], and SSDnet [76]. Also, these matrices (input matrix and filter matrix) were created using random variables. Table 4.1 illustrates this comparison for different input sizes and Figure 4.1 shows a bar chart of the same results. This experiment shows that on average our proposed convolution method is almost 10-times faster than the convolution operation in PyTorch. The point of this experiment is that both PyTorch and our proposed implementation are based on parallel programing. PyTorch is based on CUDA data parallelism, and our proposed method is based on OpenCL data parallelism.

Table 4.1 Execution Time Comparison (Our Method vs PyTorch)

Test case	Input size w*h*c	Filter size w*h*c*n	Stride	Padding	PyTorch(s)	Proposed method(s)
Test 1	224*224*3	7*7*3*64	2	2	0.872	0.091
Test 2	224*224*64	3*3*64*64	2	2	0.776	0.081
Test 3	224*224*64	1*1*64*256	1	0	1.432	0.15
Test 4	64*64*256	1*1*256*64	1	0	0.140	0.011
Test 5	64*64*256	1*1*256*128	1	0	0.232	0.025

Test 6	8*8*1024	1*1*1024*256	1	0	0.058	0.009
Test 7	112*112*64	3*3*64*64	2	2	0.254	0.021

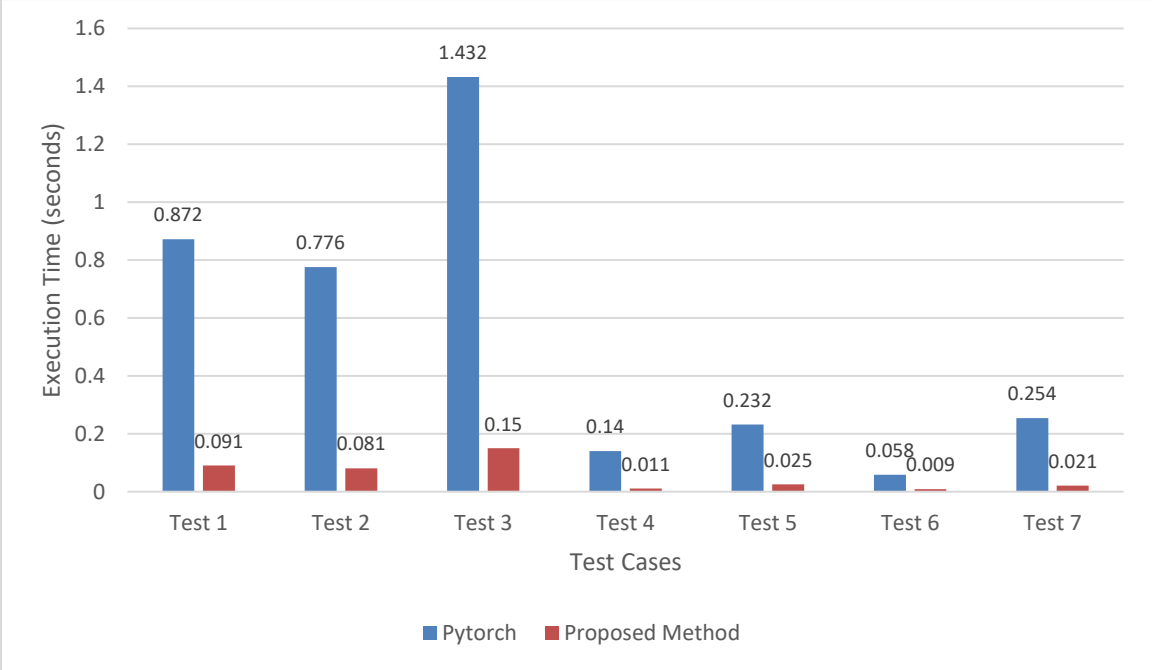


Figure 4.1 Execution Time Comparison (Our Method vs PyTorch)

We perform another experiment to determine which OpenCL kernel functions consumes the most time during the execution of the host and OpenCL application. As mentioned in Chapter 3, the OpenCL application contains three kernel functions: “expand_matrix()” function, which is used for padding the input matrix, “creat_mec()” function, which is used for generating the MEC lowered matrix; and “mul()” which is used for performing MEC multiplication between the lowered matrix and the filters. The results of this experiment can be seen in Table 4.2. The table shows the average execution time of each kernel function and required time to transfer data from the host to the OpenCL device after 100 executions. This experiment shows that almost 10% of the execution time is used only for transferring data between the host and the device. Also, approximately 10% of the time is used by the “expand_matrix()” function to pad the input matrix. As predicted, the multiplication and creation of the MEC lowered matrix needed more time

to execute, around 50% and 25%, respectively. Also, the rest of the time (2% to 5%) is used for other processing in the host software, like reading the OpenCL source code file and compiling it.

According to this experiment, when padding is zero (which means there is no need to expand the input matrix), “expand_matrix()” is not called, which results in zero present in the table.

Table 4.2 OpenCL Kernel Functions’ Execution Time

Input size w*h*c	Filter size w*h*c*n	Stride	padding	extend_matrix	creat_mec	mul	Data transfer
224*224*3	7*7*3*64	2	2	10%	25.4%	51%	11.8%
224*224*64	3*3*64*64	2	2	11.2%	25.6%	50%	11.1%
224*224*64	1*1*64*256	1	0	0%	27.5%	56%	13.8%
64*64*256	1*1*256*64	1	0	0%	30.1%	53%	14.4%
64*64*256	1*1*256*128	1	0	0%	28.3%	52%	14.2%
8*8*1024	1*1*1024*256	1	0	0%	27.5%	55%	13.8%
112*112*64	3*3*64*64	2	2	11.4%	25.1%	50%	11.2%

4.1.2 Memory Analysis

NVIDIA Jetson Nano’s 2GB of random-access memory is shared between its CPU and its GPU. For profiling the memory usage of the target device, we executed the PyTorch convolution operation and our proposed method in a loop of 100 iterations. Table 4.3 compares average memory usage after 100 executions between the two methods. Table 4.2 and its corresponding chart Figure 4.2 illustrate that our implementation used around one fifth of memory used by PyTorch full precision implementation.

Table 4.3 Memory Usage Comparison (Our Method vs PyTorch)

Test case	Input size w*h*c	Filter size w*h*c*n	Stride	padding	PyTorch (MB)	Proposed method (MB)
Test 1	224*224*3	7*7*3*64	2	2	173	34
Test 2	224*224*64	3*3*64*64	2	2	185	38
Test 3	224*224*64	1*1*64*256	1	0	232	45
Test 4	64*64*256	1*1*256*64	1	0	175	35
Test 5	64*64*256	1*1*256*128	1	0	176	35
Test 6	8*8*1024	1*1*1024*256	1	0	171	34
Test 7	112*112*64	3*3*64*64	2	2	174	34

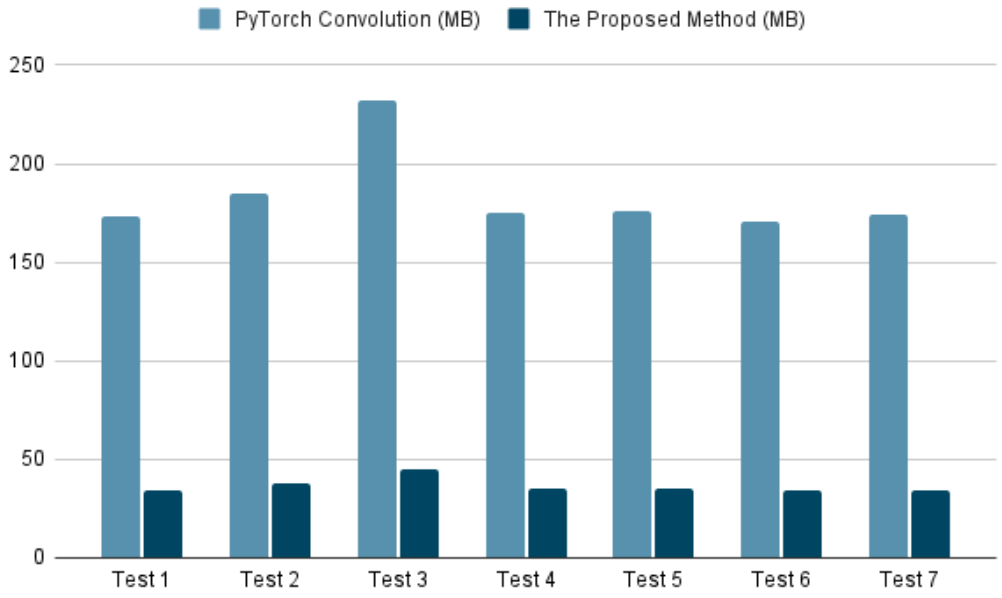


Figure 4.2 Memory Usage Comparison (Our Method vs PyTorch)

The Method as Convolution Layer

4.2 The Convolution Layer

Our second approach to testing and analyzing the proposed method is to consider it as a part of a network that is connected to other layers. To provide this type of evaluation, we must consider both large and simple networks. Depending on the neural network architecture and the purpose of the network, different metrics can be used to evaluate network accuracy. Among all evaluation metrics that have been suggested for scoring the performance of the network, we used the accuracy score. Some of these metrics are mean absolute error, mean squared error, area under curve, etc. Mean Absolute Error is the average of the difference between the Original Values and the Predicted Values. It provides a measure of how far the predictions were from the actual output. Mean Squared Error (MSE) is similar to Mean Absolute Error, the only difference being that MSE takes the average of the square of the difference between the original values and the predicted values. Area Under Curve (AUC) is one of the most widely used metrics for evaluation and is commonly used for binary classification problem. AUC of a classifier is equal to the probability that the classifier will rank a randomly chosen positive example higher than a randomly chosen negative example. Although all these metrics could help us to evaluate our models from different point of view, as we used parameter quantization as a part of the solution, we considered only accuracy as our evaluation metric. On the other word, using parameter quantization seemed to be scary in terms of accuracy drop and we needed to test it to address our concern.

The most frequently used layers in CNNs are the SoftMax layer, pooling layer, fully connected layer, ReLU layer, and convolutional layer. The SoftMax layer is typically the final output layer in a neural network that performs multi-class classification by taking a vector z of K real numbers as input and normalizing it into a probability distribution consisting of K probabilities proportional. Pooling layers are used to reduce the dimensions of the feature maps; reducing the number of parameters to learn and the amount of computation performed in the network. A fully connected layer multiplies the input by a weight matrix and then adds a bias vector. A fully connected layer can be replaced by a convolution layer. One of the possible strategies to do this is replacing a layer with a convolutional layer in which the filter size is the same as input matrix size. A

Rectified Linear Unit (ReLU) layer performs a threshold operation to each input element, where any value less than zero is set to zero.

4.2.1 Simple Network Accuracy

In the first experiment, we determine the accuracy of a simple image classification network trained using the MNIST dataset. The network used for testing the accuracy is structured as shown in Figure 4.3. As illustrated in Figure 4.3, the network contains one convolutional layer, two fully connected layers, two ReLU layers, a pooling layer, and a SoftMax layer which are connected in line. As a fully connected layer can be carried out using a convolution operation, we used our method to perform the convolution operation and fully connected layers of the network. In this experiment we trained the network with a different size of training dataset to show effect of training process on BNN accuracy as well. After training finished, we checked the learned parameters to determine β which is the hyperparameter of our proposed transfer function (as introduced in Section 3.2.3.2). After transferring the learning parameters to the binary format, we executed the model using a testing dataset. Table 4.4 illustrates the accuracy of the network on different size of training dataset. The metric we considered as the accuracy metric in this experiment is based on True Positive. True Positive-based accuracy shows the percentage of situations when the predicted class is the same as its label. Equation 4.1 shows the formulated form of this metric. According to this equation, for a image detection model if the size of training set is 500 images and the network predicts 450 cases correctly the accuracy will be 90%.

$$Accuracy = \frac{Number\ of\ correct\ predictions}{Total\ number\ of\ predicts} \times 100 \quad \textbf{(4.1)}$$

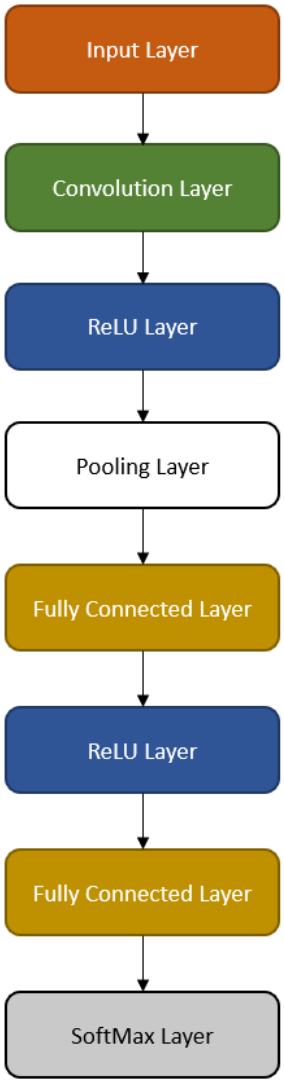


Figure 4.3 MNIST Network

Table 4.4 shows that by applying the proposed method to a simple network (MNIST-based network) the accuracy of the prediction will experience a marginal decline in comparison with full precision form of the network. Also, it shows that the amount of decrease is related to the accuracy of the original implementation (full precision version) of the network and the training process. So, for a well-trained network with 10000 images as the training data, this difference is less than 1% (from 96.7% in original implementation to 95.8% after applying our proposed method), whereas that of the same network which is trained with 6000 images as training data is almost 3% (from 90% in basic implementation to 87.3% after applying the algorithm).

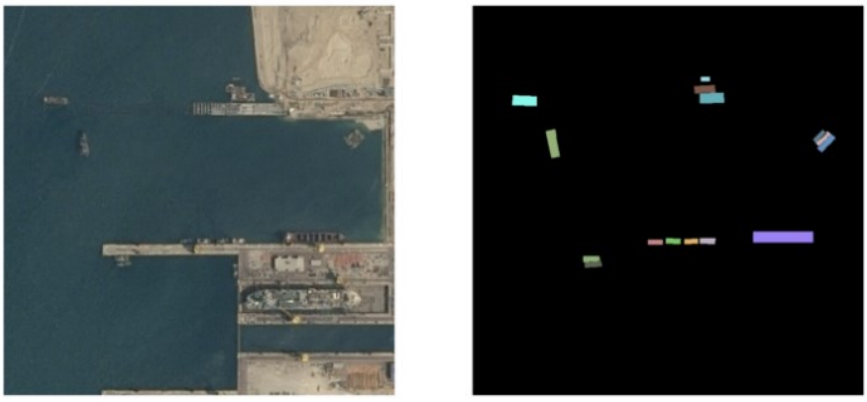
Table 4.4 MNIST Network Accuracy

Training data	Testing data	Full Precision Network Accuracy (%)	Our Method Accuracy (%)	β value
10000	2000	96.7	95.8	0
8000	2000	94.6	92.9	0.51
6000	2000	90	87.3	0.59
4000	2000	88.3	84.5	- 0.04
2000	2000	80.4	72.3	0.12
1000	2000	60	56.9	0.3
500	2000	34.6	28.2	0.8
100	2000	11.9	10.3	-0.47

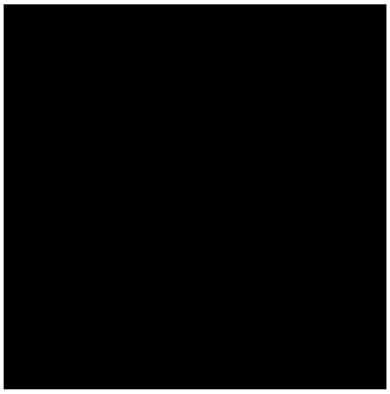
4.2.2 Complex Network Accuracy

In this third experiment, we tried to determine if the method can still be powerful even for complex networks like Mask-RCNN. First, we need to choose a library that provides a neural network implementation for different purposes. Among different options like Detectron2 [45], ImageAI [46], Gluon CV [47], Darkflow [48], etcDetectron2 is Facebook AI Research's next-generation library that provides state-of-art detection and segmentation algorithms and provides a large set of trained models available for developers. This rich set of models led us to choose Detectron2 over other libraries for our implementation. For evaluating the accuracy of a network, which is aimed at detecting or segmenting objects, two different approaches are available. In the first approach, the same as before, the accuracy of the network will be calculated by determining the percentage of true positive results of running the model on a testing data

set. In the second approach, which is a qualitative analysis, the output of the model will be determined without paying attention to the corresponding labels. In this part, we named them quantitative analysis and qualitative analysis, respectively. One of the easiest ways to evaluate a masking model is using pixel accuracy. Pixel accuracy is a qualitative analysis of a model which shows the percent of pixels in the image that are classified correctly. Although this metric is easy to calculate, it is not accurate. For example, suppose a model is aimed at segmenting ships in a satellite image. Figure 4.4a illustrates the input image, and the ground truth (what the model is supposed to segment) is illustrated in Figure 4.4b. If the model segments the image, as shown in Figure 4.4c its accuracy would be 95% which is a high value for a segmentation model even though it has detected/segmented no ships. This simple example shows that using pixel accuracy does not work for object segmentation models. A solution to the issue is using the Intersection-over Union (IoU), also known as the Jaccard Index, as an evaluation metric. IoU is the area of overlap between the predicted segmentation and the ground truth divided by the area of union between the predicted segmentation and the ground truth. Equation 4.2 demonstrates how the IoU is calculated. Figure 4.5 provides a visual representation of how IoU computes the accuracy. For using the benefits of IoU other parameters have been defined based on IoU. Average Precision (AP) is one of them. The AP is averaged over multiple IoU values [49]. In this experiment, we used AP as the metric to evaluate our proposed method.



(a) **(b)**



(c)

Figure 4.4 Illustration of Pixel Accuracy on Object Segmentation

$$IoU = \frac{Area\ of\ Overlap}{Area\ of\ Union} \tag{4.2}$$

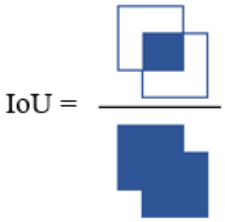


Figure 4.5 IoU Illustration

So far, we have discussed the metric that we used to evaluate the accuracy of object detection and segmentation models. To study the effect of our proposed method on the prediction accuracy of object detection and segmentation models, firstly, we started by quantitative analysis of some pretrained models after applying our proposed method. The networks we used for the study were trained by Facebook and published as a repository called detectron2_model_zoo [50]. Table 4.5 provides a comparison between different networks in terms of accuracy. The architecture of the following networks is based on Mask R-CNN. Also, all the networks have been trained using the COCO [51] dataset for detecting and segmenting vehicles in a scene. Each network's backbone is provided in the table. We mentioned another training parameter which is epoch, a hyperparameter, which

is one entire transit of the training data through the algorithm measured in training steps. For example, in the first row the epoch is 100 which means, the training set has been passed forward and backward 100 times through the network during the training process. In this experiment, one epoch consists of training on 118,000 COCO images. As the training and validation dataset is generally split in an 80 to 20 ratio, we considered 29,500 images as validation data. Also, all hyperparameters of the networks are as the same as the default values of the hyperparameters for the networks in Detectron2 implementation. The only hyperparameter that has been changed in this experiment is the value of the epoch.

Table 4.5 Comparison Between Different Networks in Terms of Accuracy

Name	Epoch	Network Backbone	Original version accuracy (AP)	Accuracy after applying our method (AP)
R50-FPN	100	ResNet 50	40.3	34.1
R50-FPN	200	ResNet 50	41.7	35
R50-FPN	400	ResNet 50	42.5	35.2
R101-FPN	100	ResNet 101	41.6	34.9
R101-FPN	200	ResNet 101	43.1	36.3
R101-FPN	400	ResNet 101	43.7	36.8

According to Figure 4.6 and the definition of AP, 40% accuracy means, for example, if the masked area is 140×100 pixels, it has 100×80 overlap with the union area (in this case union area is 20000 pixels). Figure 4.6 shows the calculations. In fact, 40% of AP in this example means the network masked $4/7$ of the object correctly.

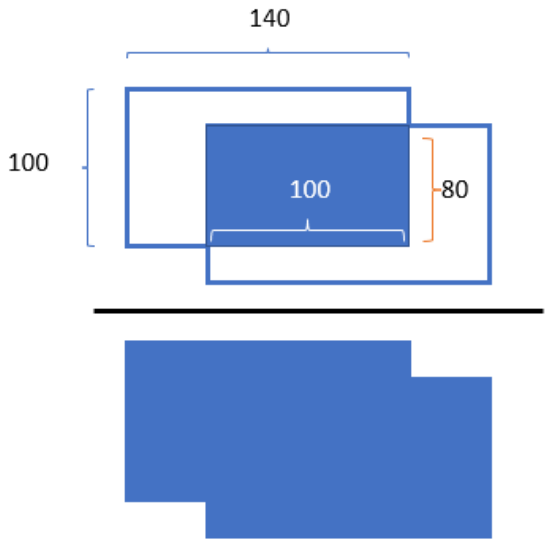


Figure 4.6 Example of AP

The same calculations show that if the AP is 34%, it means the network masks $^{68}/_{134}$ of the object correctly. These calculations of the first row of Table 4.5 can be extended for the other rows as well. As the table demonstrates, by using our method, we only lose almost 10% of object masking accuracy in comparison with the full precision version of the network.

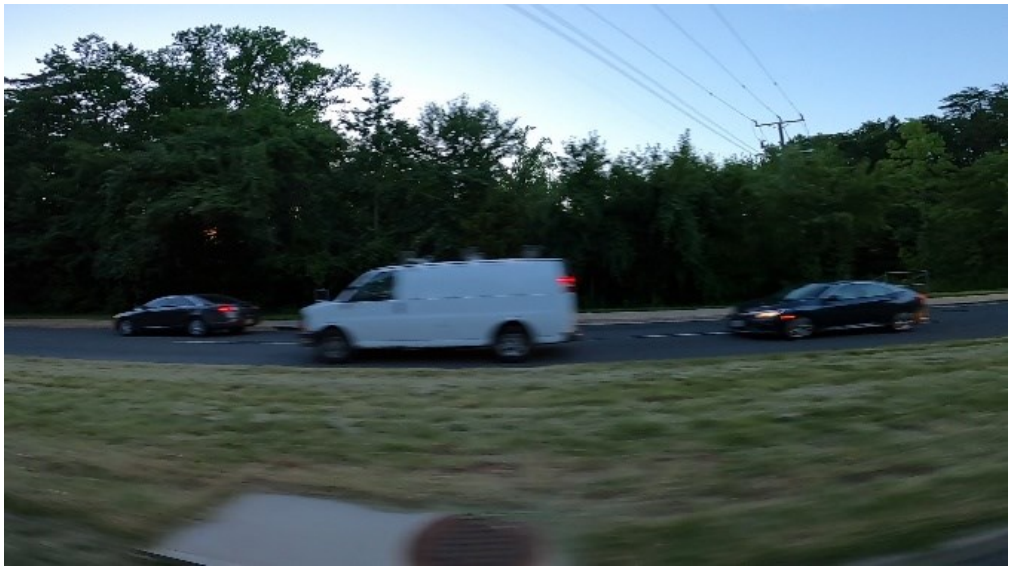
To provide a visual sense of the accuracy of the models after using the method, we provided some examples of the output of the models. Figures 4.7 to 4.13 contain three parts. Part (a) is the input image, Part (b) illustrates the output of the original model using PyTorch, and Part (c) demonstrates the model's output using our method for the same input. These images are some images taken outside the COCO dataset.

(a)

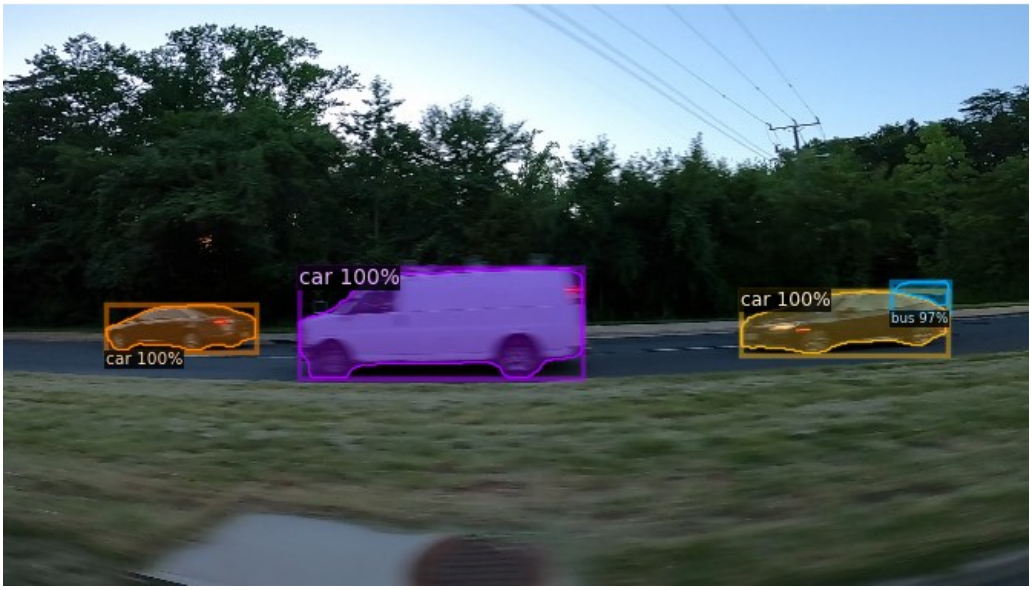
(b)

(c)

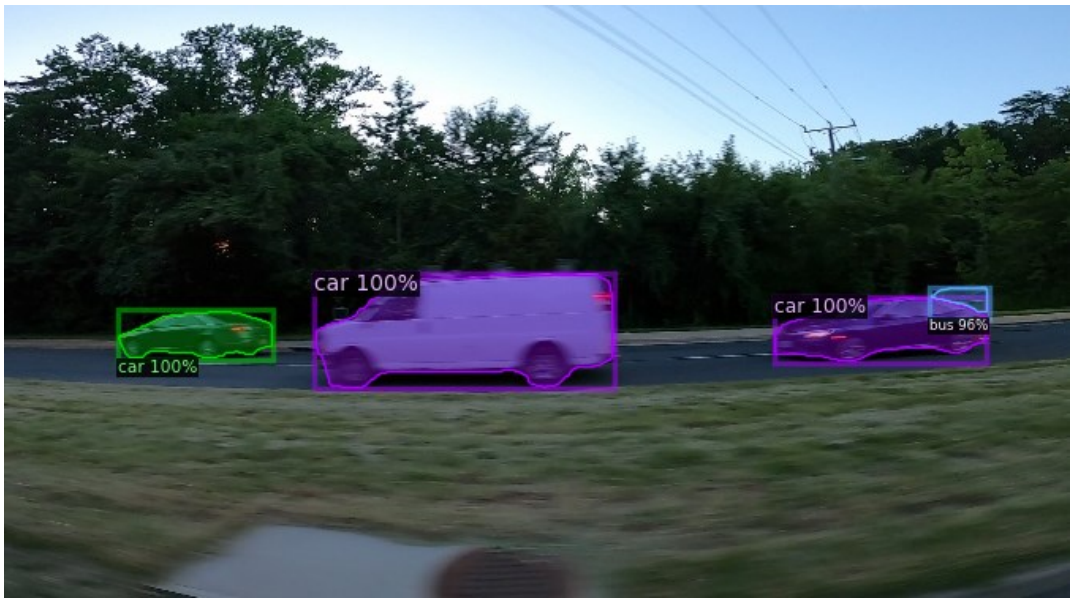
Figure 4.7 Output of R50-FPN (Epoch 100) Model on Test Data



(a)



(b)



(c)

Figure 4.8 Output of R50-FPN (Epoch 100) Model on Test Data

(a)

(b)

(c)

Figure 4.9 Output of R50-FPN (Epoch 200) Model on Test Data

(a)

(b)

(c)

Figure 4.10 Output of R50-FPN (Epoch 400) Model on Test Data

(a)

(b)

(c)

Figure 4.11 Output of R101-FPN (Epoch 100) Model on Test Data

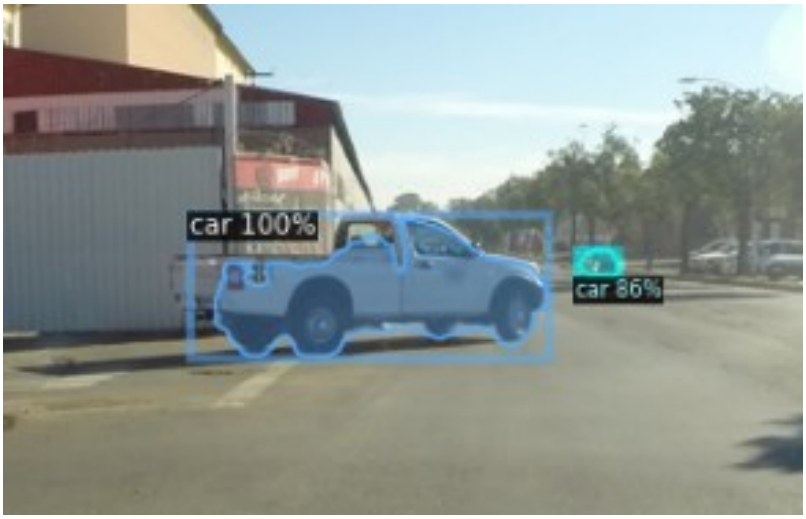
(a)

(b)

(c)

Figure 4.12 Output of R101-FPN (Epoch 200) Model on Test Data

(a)



(b)

(c)

Figure 4.13 Output of R101-FPN (Epoch 400) Model on Test Data

Chapter 5:

Conclusion

In the previous chapter, we have attempted to evaluate our proposed method by testing it under various conditions. We showed that if we run our proposed method as a convolution operation it can be up to 10-times faster and 80% more memory efficient than the PyTorch standard implementation for the convolutional operations. Also, we considered our method as a convolution operation in various neural networks and checked its effect on the networks' accuracy. This chapter is split into two sections, the first covers an overall conclusion about the method and the results of the experiments: the second covers future work.

5.1 The Method

For improving the memory usage and runtime of CNNs we provide a solution which is an aggregation and update of three different approaches. We used the MEC to perform the convolutional operation and combined this with binarization as a parameter quantization method to reduce memory usage and improve the execution time of CNNs. We provided a technique to train a binary network and then used an OpenCL-based implementation for the solution. Among all these steps, MEC and software implementation have no effect on the accuracy of CNNs. What changes the accuracy of the networks is the parameter quantization part of the solution and tested the effect of that quantization on the networks' accuracy (simple and complex networks). Although the experiments showed good results in terms of accuracy, using binarization is not the final solution to improve the efficiency of all types of networks. For example, networks aimed at performing a variety of tasks would not find that binarization is not an absolute solution - if a network is trained by a huge dataset to detect different unrelated objects like cars, pens, laptops, books, etc. this method is not useful, and the reason is that the networks training is distributed across a wider range of various images and therefore each digit has more

influence on the final accuracy and thus the full precision parameters carry more weight.. Overall, this method can help us to improve the efficiency in terms of memory usage and runtime for simple networks or complex networks that are limited to performing specific tasks like only detecting cars in images. The solution to this issue will be covered in the next section.

5.2 Future work

There are other possible solutions that we can use to combine with a current solution to help us improve the functionality of the CNNs.

A first potential solution we would propose using a structured pruning in the training process right before transferring knowledge by the transfer function. If the pruning is too aggressive it may lead us to lose a large set of learned knowledge and therefore, we need to consider the fact that the next step would be parameter quantization.

The second possible solution to improve the efficiency would be to find a binary solution and binary representation for other operations in the networks. Although convolutional layers are the main resource users in a CNN, it is not the only layer. Finding a similar solution for the other networks would be a potential solution.

The last possible solution which is represented here could be the solution to the raised issue in Section 5.1, i.e., a major reduction in the accuracy of very complex networks which are trained to do different tasks. In these networks, using the proposed method is not useful. Instead, we would use selection binarization, whereby before applying the method, we would study the network and determine the convolutional layers which used more resources than other convolutional layers and apply our method only to those layers.

References

[1] Xygkis, Athanasios, Dimitrios Soudris, Lazaros Papadopoulos, Sofiane Yous, and David Moloney. 2018. “Efficient Winograd-Based Convolution Kernel Implementation on Edge Devices.” 2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC), June.

[2] Wang, Zi, Chengcheng Li, and Xiangyang Wang. "Convolutional Neural Network Pruning with Structural Redundancy Reduction." In Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition, pp. 14913-14922. 2021.

[3] Blalock, Davis, Jose Javier Gonzalez Ortiz, Jonathan Frankle, and John Guttag. "What Is The State Of Neural Network Pruning?." Proceedings of machine learning and systems 2 (2020): 129-146

[4] Li, Hao, Asim Kadav, Igor Durdanovic, Hanan Samet, and Hans Peter Graf. "Pruning Filters For Efficient Convnets." arXiv preprint arXiv:1608.08710 (2016).

[5] Ma, Xiaolong, Sheng Lin, Shaokai Ye, Zhezhi He, Linfeng Zhang, Geng Yuan, Sia Huat Tan et al. "Non-Structured DNN Weight Pruning--Is It Beneficial in Any Platform?." IEEE Transactions on Neural Networks and Learning Systems (2021).

[6] Han, Song, Jeff Pool, John Tran, and William Dally. "Learning Both Weights And Connections For Efficient Neural Network." Advances in neural information processing systems 28 (2015)

[7] Chang, Jing, and Jin Sha. "An Efficient Implementation Of 2D Convolution In CNN." IEICE Electronics Express (2016): 13-20161134.

[8] Chellapilla, Kumar, Sidd Puri, and Patrice Simard. "High Performance Convolutional Neural Networks For Document Processing." In Tenth international workshop on frontiers in handwriting recognition. Suvisoft, 2006.

[9] Cong, Jason, and Bingjun Xiao. "Minimizing Computation In Convolutional Neural Networks." In International conference on artificial neural networks, pp. 281-290. Springer, Cham, 2014.

[10] Cho, Minsik, and Daniel Brand. "MEC: Memory-Efficient Convolution For Deep Neural Network." In International Conference on Machine Learning, pp. 815-824. PMLR, 2017.

[11] Krizhevsky, Alex, Ilya Sutskever, and Geoffrey E. Hinton. "Imagenet Classification With Deep Convolutional Neural Networks." Advances in neural information processing systems 25 (2012).

[12] Abtahi, Tahmid, Colin Shea, Amey Kulkarni, and Tinoosh Mohsenin. "Accelerating Convolutional Neural Network With FFT On Embedded Hardware." IEEE Transactions on Very Large Scale Integration (VLSI) Systems 26, no. 9 (2018): 1737-1749.

[13] "Jetson NANO 2GB" Jetson NANO 2GB. [Online]. Available: https://www.nvidia.com/en-us/autonomous-machines/embedded-systems/jetson-nano/education-projects/. [Accessed: 19-Apr-2022].

[14] Highlander, Tyler, and Andres Rodriguez. "Very Efficient Training Of Convolutional Neural Networks Using Fast Fourier Transform And Overlap-And-Add." arXiv preprint arXiv:1601.06815 (2016).

[15] Park, Hyunsun, Dongyoung Kim, Junwhan Ahn, and Sungjoo Yoo. "Zero And Data Reuse-Aware Fast Convolution For Deep Neural Networks On GPU." In Proceedings of the Eleventh IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis, pp. 1-10. 2016.

[16] Winograd, Shmuel. 1980. Arithmetic Complexity of Computations. Philadelphia: Siam.

[17] Goyal, Rishabh, Joaquin Vanschoren, Victor Van Acht, and Stephan Nijssen. "Fixed-point Quantization of Convolutional Neural Networks for Quantized Inference on Embedded Platforms." arXiv preprint arXiv:2102.02147 (2021).

[18] Liu, Zechun, Wenhan Luo, Baoyuan Wu, Xin Yang, Wei Liu, and Kwang-Ting Cheng. "Bi-Real Net: Binarizing Deep Network Towards Real-Network Performance." International Journal of Computer Vision 128, no. 1 (2020): 202-219.

[19] Courbariaux, Matthieu, Yoshua Bengio, and Jean-Pierre David. "Binaryconnect: Training Deep Neural Networks With Binary Weights During Propagations." Advances in neural information processing systems 28 (2015).

[20] Qin, Haotong, Ruihao Gong, Xianglong Liu, Xiao Bai, Jingkuan Song, and Nicu Sebe. "Binary Neural Networks: A Survey." Pattern Recognition 105 (2020): 107281.

[21] Bengio, Yoshua, Nicholas Léonard, and Aaron Courville. "Estimating Or Propagating Gradients Through Stochastic Neurons For Conditional Computation." arXiv preprint arXiv:1308.3432 (2013).

[22] Liu, Zechun, Baoyuan Wu, Wenhan Luo, Xin Yang, Wei Liu, and Kwang-Ting Cheng. "Bi-Real Net: Enhancing The Performance Of 1-Bit Cnns With Improved

Representational Capability And Advanced Training Algorithm." In Proceedings of the European conference on computer vision (ECCV), pp. 722-737. 2018.

[23] Liu, Chunlei, Wenrui Ding, Xin Xia, Baochang Zhang, Jiaxin Gu, Jianzhuang Liu, Rongrong Ji, and David Doermann. "Circulant Binary Convolutional Networks: Enhancing The Performance Of 1-Bit Dcnns With Circulant Back Propagation." In Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition, pp. 2691-2699. 2019.

[24] Wang, Ziwei, Jiwen Lu, Ziyi Wu, and Jie Zhou. "Learning Efficient Binarized Object Detectors With Information Compression." IEEE Transactions on Pattern Analysis and Machine Intelligence (2021).

[25] Wang, Yizhi, Jun Lin, and Zhongfeng Wang. "An Energy-Efficient Architecture For Binary Weight Convolutional Neural Networks." IEEE Transactions on Very Large Scale Integration (VLSI) Systems 26, no. 2 (2017): 280-293.

[26] Galloway, Angus, Graham W. Taylor, and Medhat Moussa. "Attacking Binarized Neural Networks." arXiv preprint arXiv:1711.00449 (2017).

[27] Vanhoucke, Vincent, Andrew Senior, and Mark Z. Mao. "Improving The Speed Of Neural Networks On Cpus." (2011).

[28] Krishnamoorthi, Raghuraman. "Quantizing Deep Convolutional Networks For Efficient Inference: A Whitepaper." arXiv preprint arXiv:1806.08342 (2018).

[29] Jain, Sambhav, Albert Gural, Michael Wu, and Chris Dick. "Trained Quantization Thresholds For Accurate And Efficient Fixed-Point Inference Of Deep Neural Networks." Proceedings of Machine Learning and Systems 2 (2020): 112-128.

[30] Wu, Hao, Patrick Judd, Xiaojie Zhang, Mikhail Isaev, and Paulius Micikevicius. "Integer Quantization For Deep Learning Inference: Principles And Empirical Evaluation." arXiv preprint arXiv:2004.09602 (2020).

[31] Baskin, Chaim, Natan Liss, Eli Schwartz, Evgenii Zheltonozhskii, Raja Giryes, Alex M. Bronstein, and Avi Mendelson. "Uniq: Uniform Noise Injection For Non-Uniform Quantization Of Neural Networks." ACM Transactions on Computer Systems (TOCS) 37, no. 1-4 (2021): 1-15.

[32] Rastegari, Mohammad, Vicente Ordonez, Joseph Redmon, and Ali Farhadi. "Xnor-Net: Imagenet Classification Using Binary Convolutional Neural Networks." In European conference on computer vision, pp. 525-542. Springer, Cham, 2016.

[33] Guo, Yunhui. "A Survey On Methods And Theories Of Quantized Neural Networks." arXiv preprint arXiv:1808.04752 (2018).

[34] Wu, Shuang, Guoqi Li, Feng Chen, and Luping Shi. "Training And Inference With Integers In Deep Neural Networks." arXiv preprint arXiv:1802.04680 (2018).

[35] Cai, Zhaowei, Xiaodong He, Jian Sun, and Nuno Vasconcelos. "Deep Learning With Low Precision By Half-Wave Gaussian Quantization." In Proceedings of the IEEE conference on computer vision and pattern recognition, pp. 5918-5926. 2017.

[36] Lin, Darryl D., and Sachin S. Talathi. "Overcoming Challenges In Fixed Point Training Of Deep Convolutional Networks." arXiv preprint arXiv:1607.02241 (2016).

[37] Lin, Xiaofan, Cong Zhao, and Wei Pan. "Towards Accurate Binary Convolutional Neural Network." Advances in neural information processing systems 30 (2017).

[38] Misha Denil, Babak Shakibi, Laurent Dinh, Nando De Freitas, et al. Predicting parameters in deep learning. In Advances in neural information processing systems, pages 2148–2156, 2013.

[39] Molchanov, Dmitry, Arsenii Ashukha, and Dmitry Vetrov. "Variational Dropout Sparsifies Deep Neural Networks." In International Conference on Machine Learning, pp. 2498-2507. PMLR, 2017.

[40] Srivastava, Nitish, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. "Dropout: A Simple Way To Prevent Neural Networks From Overfitting." The journal of machine learning research 15, no. 1 (2014): 1929-1958.

[41] Esser, Steve K., Rathinakumar Appuswamy, Paul Merolla, John V. Arthur, and Dharmendra S. Modha. "Backpropagation For Energy-Efficient Neuromorphic Computing." Advances in neural information processing systems 28 (2015).

[42] Anderson, Alexander G., and Cory P. Berg. "The High-Dimensional Geometry Of Binary Neural Networks." arXiv preprint arXiv:1705.07199 (2017).

[43] "OpenCL - the Open Standard for Parallel Programming of Heterogeneous Systems." 2013. The Khronos Group. July 21, 2013. https://www.khronos.org/opencl/.

[44] "Documentation Portal." n.d. Docs.xilinx.com. Accessed April 16, 2022. https://docs.xilinx.com/r/en-US/ug1393-vitis-application-acceleration/Getting-Started-with-Vitis.

[45] "Facebookresearch/Detectron2." 2020. GitHub. May 26, 2020. https://github.com/facebookresearch/detectron2.

[46] OLAFENWA, MOSES. 2020. "OlafenwaMoses/ImageAI." GitHub. April 27, 2020. https://github.com/OlafenwaMoses/ImageAI.

[47] "Gluon CV Toolkit." 2022. GitHub. April 15, 2022. https://github.com/dmlc/gluon-cv.
[48] Trieu. 2022. "Thtrieu/Darkflow." GitHub. April 12, 2022. https://github.com/thtrieu/darkflow.

[49] "COCO - Common Objects in Context." n.d. Cocodataset.org. https://cocodataset.org/#detection-eval.

[50] "Facebookresearch/Detectron2." 2022. GitHub. April 15, 2022. https://github.com/facebookresearch/detectron2/blob/main/MODEL_ZOO.md.

[51] "COCO - Common Objects in Context." n.d. Cocodataset.org. https://cocodataset.org/#home.

[52] "GEMM - General Matrix Multiplication - Simplified Version." n.d. Www.ibm.com. Accessed April 17, 2022. https://www.ibm.com/docs/en/netezza?topic=operations-gemm-general-matrix-multiplication-simplified-version.

[53] Oh, Sangyun, Hyeonuk Sim, Sugil Lee, and Jongeun Lee. "Automated Log-Scale Quantization For Low-Cost Deep Neural Networks." In Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition, pp. 742-751. 2021.

[54] Ding, Ruizhou, Zeye Liu, Ting-Wu Chin, Diana Marculescu, and Ronald D. Blanton. "Flightnns: Lightweight Quantized Deep Neural Networks For Fast And Accurate Inference." In Proceedings of the 56th Annual Design Automation Conference 2019, pp. 1-6. 2019.

[55] "MNIST Handwritten Digit Database, Yann LeCun, Corinna Cortes and Chris Burges." 2009. Lecun.com. 2009. http://yann.lecun.com/exdb/mnist/.

[56] He, Kaiming, Georgia Gkioxari, Piotr Dollár, and Ross Girshick. "MASK R-CNN." In Proceedings of the IEEE international conference on computer vision, pp. 2961-2969. 2017.

[57] Hamid Sarbazi-Azad. 2017. Advances in GPU Research and Practice. Cambridge, Ma: Morgan Kaufmann

[58] Bouhali, Noureddine, Hamza Ouarnoughi, Smail Niar, and Abdessamad Ait El Cadi. "Execution Time Modeling for CNN Inference on Embedded GPUs." In Proceedings of the 2021 Drone Systems Engineering and Rapid Simulation and Performance Evaluation: Methods and Tools Proceedings, pp. 59-65. 2021.

[59] Tessier, Hugo. 2021. "Neural Network Pruning 101." Medium. September 13, 2021. https://towardsdatascience.com/neural-network-pruning-101-af816aaea61.

[60] Krizhevsky, Alex, Ilya Sutskever, and Geoffrey E. Hinton. "Imagenet Classification With Deep Convolutional Neural Networks." Advances in neural information processing systems 25 (2012).

[61] “ImageNet.” n.d. Image-Net.org. https://image-net.org/.

[62] Hammad, Issam, and Kamal El-Sankary. "Impact Of Approximate Multipliers On VGG Deep Learning Network." IEEE Access 6 (2018): 60438-60444.

[63] Fung, Vincent. 2017. “An Overview of ResNet and Its Variants.” Towards Data Science. Towards Data Science. July 15, 2017. https://towardsdatascience.com/an-overview-of-resnet-and-its-variants-5281e2f56035.

[64] Krizhevsky, Alex. 2009. “CIFAR-10 and CIFAR-100 Datasets.” Toronto.edu. 2009. https://www.cs.toronto.edu/~kriz/cifar.html.

[65] Ren, Shaoqing, Kaiming He, Ross Girshick, and Jian Sun. "Faster R-CNN: Towards real-time object detection with region proposal networks." Advances in neural information processing systems 28 (2015).

[66] Wikipedia Contributors. 2019. “Cooley–Tukey FFT Algorithm.” Wikipedia. Wikimedia Foundation. September 17, 2019. https://en.wikipedia.org/wiki/Cooley%E2%80%93Tukey_FFT_algorithm.

[67] Howard, Andrew G., Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. "Mobilenets: Efficient Convolutional Neural Networks For Mobile Vision Applications." arXiv preprint arXiv:1704.04861 (2017).

[68] Alake, Richmond. 2020. “Deep Learning: GoogLeNet Explained.” Medium. December 23, 2020. https://towardsdatascience.com/deep-learning-googlenet-explained-de8861c82765.

[69] He, Kaiming, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. "Deep Residual Learning For Image Recognition." In Proceedings of the IEEE conference on computer vision and pattern recognition, pp. 770-778. 2016.

[70] “Attacking Machine Learning with Adversarial Examples.” 2017. OpenAI. February 24, 2017. https://openai.com/blog/adversarial-example-research/.

[71] “CUDA Toolkit.” 2013. NVIDIA Developer. July 2, 2013. https://developer.nvidia.com/cuda-toolkit.

[72] “What Is Superword Level Parallelism?” 2021. Webopedia. May 5, 2021. https://www.webopedia.com/definitions/superword-level-parallelism/.

[73] “Intel Edge Technology and Solutions.” n.d. Intel. Accessed April 19, 2022. https://www.intel.ca/content/www/ca/en/edge-computing/overview.html?cid=psm&source=google-ads&campid=2022_q2_ccg_ca_dpgcspa_dpgcspa2_E&G-AI-Brand&content=text-ads_en.

[74] “Basic CNN Architecture: Explaining 5 Layers of Convolutional Neural Network.” 2020. UpGrad Blog. December 7, 2020. https://www.upgrad.com/blog/basic-cnn-architecture/#:~:text=There%20are%20three%20types%20of.

[75] “PyTorch,” PyTorch. [Online]. Available: https://PyTorch.org/. [Accessed: 19-Apr-2022].

[76] Liu, Wei, Dragomir Anguelov, Dumitru Erhan, Christian Szegedy, Scott Reed, Cheng-Yang Fu, and Alexander C. Berg. "Ssd: Single Shot Multibox Detector." In European conference on computer vision, pp. 21-37. Springer, Cham, 2016.

[77] “OpenCL Programming Guide — ROCm 4.5.0 Documentation.” n.d. Rocmdocs.amd.com. Accessed April 19, 2022. https://rocmdocs.amd.com/en/latest/Programming_Guides/Opencl-programming-guide.html.

[78] Zhou, Yangjie, Mengtian Yang, Cong Guo, Jingwen Leng, Yun Liang, Quan Chen, Minyi Guo, and Yuhao Zhu. "Characterizing and Demystifying the Implicit Convolution Algorithm on Commercial Matrix-Multiplication Accelerators." In 2021 IEEE International Symposium on Workload Characterization (IISWC), pp. 214-225. IEEE, 2021.

[79] Zhou, Yangjie, Mengtian Yang, Cong Guo, Jingwen Leng, Yun Liang, Quan Chen, Minyi Guo, and Yuhao Zhu. "Characterizing and Demystifying the Implicit Convolution Algorithm on Commercial Matrix-Multiplication Accelerators." In 2021 IEEE International Symposium on Workload Characterization (IISWC), pp. 214-225. IEEE, 2021.

[80] Pardhi, Prachi, Kiran Yadav, Siddhansh Shrivastav, Satya Prakash Sahu, and Deepak Kumar Dewangan. "Vehicle Motion Prediction For Autonomous Navigation System Using 3 Dimensional Convolutional Neural Network." In 2021 5th International Conference on Computing Methodologies and Communication (ICCMC), pp. 1322-1329. IEEE, 2021.

[81] Gholami, Amir, Sehoon Kim, Zhen Dong, Zhewei Yao, Michael W. Mahoney, and Kurt Keutzer. "A Survey Of Quantization Methods For Efficient Neural Network Inference." arXiv preprint arXiv:2103.13630 (2021).

[82] Liu, Zechun, Zhiqiang Shen, Shichao Li, Koen Helwegen, Dong Huang, and Kwang-Ting Cheng. "How Do Adam And Training Strategies Help Bnns Optimization." In International Conference on Machine Learning, pp. 6936-6946. PMLR, 2021.

[83] Laydevant, Jérémie, Maxence Ernoult, Damien Querlioz, and Julie Grollier. "Training Dynamical Binary Neural Networks With Equilibrium Propagation." In Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition, pp. 4640-4649. 2021.

[84] Ghimire, Deepak, Dayoung Kil, and Seong-heum Kim. "A Survey on Efficient Convolutional Neural Networks and Hardware Acceleration." Electronics 11, no. 6 (2022): 945.

[85] Xiao, Lechao, Yasaman Bahri, Jascha Sohl-Dickstein, Samuel Schoenholz, and Jeffrey Pennington. "Dynamical Isometry And A Mean Field Theory Of Cnns: How To Train 10,000-Layer Vanilla Convolutional Neural Networks." In International Conference on Machine Learning, pp. 5393-5402. PMLR, 2018.

[86] “Tensorflow,” Tensorflow. [Online]. Available: https://www.tensorflow.org/. [Accessed: 19-Apr-2022].