Study of Implementation of CNN on Low-power Platform for Smart Traffic Optimization

by

Zhizhou Li

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Applied Science
in
Electrical and Computer Engineering

Waterloo, Ontario, Canada, 2017

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

Accompanied with the rise of smart city and the development of IoT (Internet of Things), people are looking forward to monitoring and regulating the traffic in a smarter way. Since the deep neural network has shown its great value in vehicle detection area, people may wonder what kind of impact would be brought by the combination of IoT and deep learning techniques. In this work, an exploration of implementation of CNN (convolutional neural network) on low-power platform for smart traffic optimization has been conducted. During the research, a new optimization approach, which aims at S-CNN (Sparse Convolutional Neural Network) optimization from architecture level, has been proposed; and outstanding performance has been obtained when compared to mainstream deep learning frameworks, such as Tensorflow. In the experiments, the new proposed S-CNN optimization approach is as twice fast as Tensorflow on 94% sparse model and becomes 5 times faster on 98% sparse model. Besides, the author also verified the feasibility of real-time CNN implementation on ARM platform and Jetson TX1 embedded system, which reveals the shortage of computational resource on ARM platform and the potential of Jetson series to become the low-power platform for CNN implementation.

Acknowledgements

First and foremost, I would like to show my deepest gratitude to my supervisors, Dr. Kshirasagar Naik and Dr. Justin Eichel. They are respectable, responsible and resourceful scholars, who have provided me with valuable guidance in every stage of the writing of this thesis. Without their enlightening instruction, impressive kindness and patience, I could not have completed my thesis.

I shall also extend my thanks to ECE Graduate Office and the staff for their kindness, help and sponsorship. I would also like to thank all my teachers who have helped me to develop the fundamental and essential academic competence.

Last but not least, I would like to thank all my friends for their encouragement and support.

Dedication

I dedicate this thesis to my parents, who gave me the first book in my life.

Table of Contents

List of Tables

List of Figures

List of Abbreviations

API Application Programming Interface 17

ARM Advanced RISC Machine 13, 15, 17, 21, 22, 29–32, 36, 39

BLAS Basic Linear Algebra Subprograms 17

CNN Convolutional Neural Network 3–5, 7, 9, 12, 21, 24, 27, 32, 34, 41, 42

CPU Central Processing Unit 11–13, 17, 29–32, 34, 36, 39, 42

FLOPS Floating-point Operations Per Second 9

GPS Global Positioning System 2

GPU Graphics Processing Unit 11–13, 15, 17, 29–31, 34, 36, 39, 42

IoT Internet of Things 1

NEMA National Electrical Manufacturers Association 19

NN Neural Network 5

ReLU Rectified Linear Unit 7, 10

RISC Reduced Instruction Set Computing 13

SIMD Single Instruction Multiple Data 13

STL Standard Template Library 22

TPU Tensor Processing Unit 42

VGG Visual Geometry Group 7, 9, 10

Nomenclature

S-CNN A type of convolutional neural network, whose parameters contain a lot of zeros. 9, 10, 21, 22, 24, 26, 27, 30–32, 40, 42

Chapter 1

Introduction

1.1 Motivation

"Smart City" is a recently invented terminology in Internet of Things (IoT) area. It utilizes the technology of IoT and other modern technique to implement informatilization, industrialisation and urbanization. According to the evaluation criterion of University of Vienna[13], a smart city should include the following six characteristics: smart economy, smart people, smart governance, smart mobility, smart environment and smart living. Figure 1.1 shows the concept of smart city intuitively. The smart city, as an organic whole, consists of these indispensable components.

Since we have already known that smart mobility could help improve the intelligence of a city, we may also want to know how we can make the mobility smarter. For better analysis, it is useful to divide the whole traffic system into two parts: traffic on the road and traffic off the road. Table 1.1 shows the difference between these two types of traffic. Compared to the off-road traffic, the on-road traffic is the bottleneck of smart mobility based on its features. First, the on-road traffic consists of various land vehicles, which means it is hard to optimize the traffic based on type and size of vehicles. Second, the controllability of on-road traffic is quite low because current traffic regulation system lacks real-time compulsory control. For example, a local driver may still drive on his daily route towards home even though he has already knew there may be congestion around the rush hour today. Lastly, the biggest problem of optimization for on-road traffic is its high burstiness. The burstiness comprises two parts: the predictable one and the unpredictable one. The predictable burst could be mostly avoided by advanced notification and smart route planning, but the unpredictable burst is hard to keep track immediately and regulate

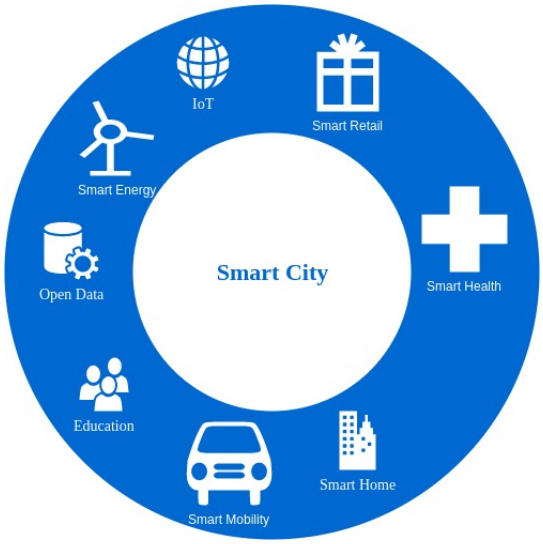


Figure 1.1: The Basic Components of Smart City

Table 1.1: Comparison of On-road Traffic and Off-road Traffic

Traffic Type	On-road Traffic	Off-road Traffic
Composing Unit	Various vehicles (most of them are private cars), pedestrians, cyclists, etc.	Large-scale transportation system, such as railways and airlines.
Controllability	Low	High
Burstiness	High	Low

correspondingly. Although with such difficulty, it is without doubt that people are looking forward to optimization for current on-road traffic, such as reducing congestion and real-time road situation monitoring.

By summarizing the problems mentioned above, to optimize on-road traffic requires real-time and widely distributed measurement. Only with such dynamic and detailed information, people are allowed to develop appropriate algorithms for the smart mobility. This kind of measurement is to know the road situation, such as the location of vehicles, pedestrians and cyclists, so that the system could analyze the congestion situation and plan as a whole. There are many methods to locate a car, like Global Positioning System (GPS) and vehicle detection via computer vision technique. The GPS measurement is limited by the availability and accuracy of GPS equipment in the cars and cannot cover widely enough to support large-scale traffic optimization. Therefore, the vehicle detection

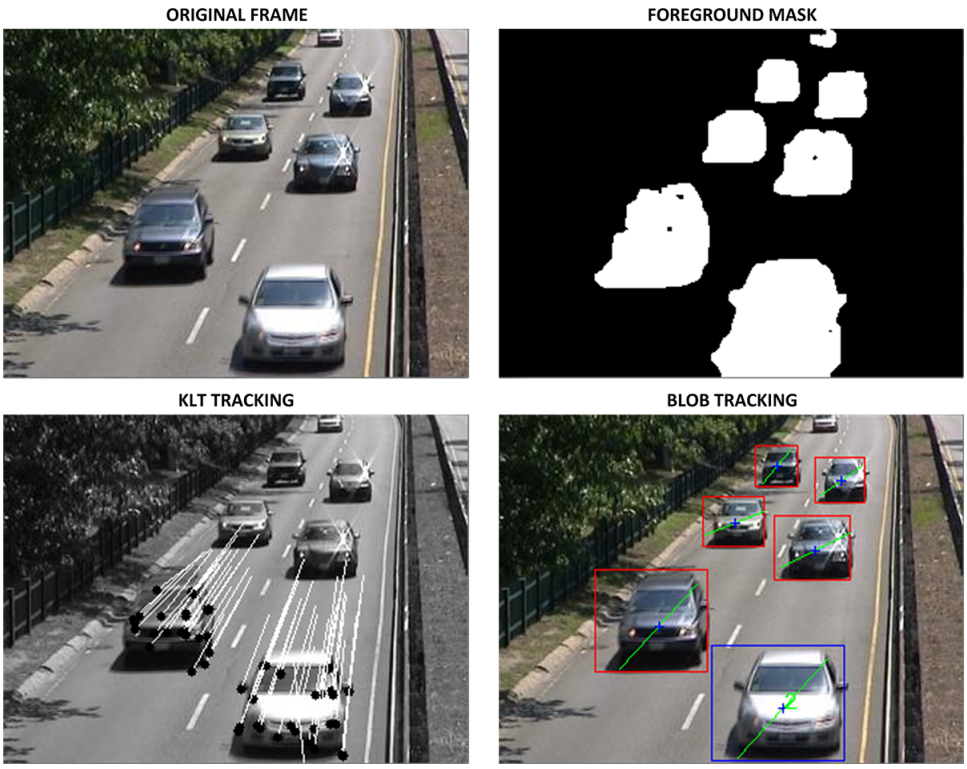


Figure 1.2: Vehicle Detection via convolutional neural networks

method using computer vision technique could be the most suitable way to obtain expected road situation for now.

The vehicle detection requires computers to recognize the vehicles and related contents in a picture. The state-of-art method of pattern recognition in computer vision is using Convolutional Neural Network (CNN), and it could achieve satisfactory results after enough training and careful deployment. Figure 1.2 shows several vehicle detection effects using CNN.

However, the normal vehicle detection system is power-consuming, which means it cannot be deployed widely and become self-powered. Hence, how to realize a low-power vehicle detection system becomes a popular topic.

1.2 Thesis Architecture

This thesis focuses on low-power vehicle detection system design and tries to figure out a feasible solution to improve CNN implementation on such low-power platforms. The general idea of the thesis is to test several software-hardware combination and sort out the most feasible solution among them. The following chapters display the research progress and results.

Chapter 2 introduces the background knowledge needed for this thesis. Chapter 3 proposes the research problem and its further analysis. Chapter 4 explains the experiment setup details and the solution strategy. Chapter 5 reports the experiment results with corresponding explanation. Chapter 6 shows conclusions based on previous experiments and related discussion. The last chapter illustrates some future work according to current research and exploration.

Chapter 2

Background

This chapter introduces the background knowledge needed for understanding this thesis.

2.1 Convolutional Neural Network

Convolutional Neural Networks (CNNs) is a variant of ordinary Neural Networks. LeNet[12] is one of the earliest CNNs, which has boosted the development of Deep Learning[11]. Firstly, it is necessary to understand what is the 2D convolution in computer vision area. Figure 2.1 demonstrates a simple example of 2D convolution. In this example, the convolution operation could be seen as the process of feature extraction[5]. At the beginning, when people tried to apply machine learning algorithms (for example, regression models) to solving image recognition problem, matrix multiplication is used between each layers to simulate the connection of biological neurons. Instead of using large-scale matrix operations, CNNs use multiple small-scale matrix convolutions for feature detection and abstraction. The benefit of replacing matrix multiplication with convolution is obvious. The number of trainable parameters of the entire network becomes much smaller compared to ordinary Neural Network (NN).

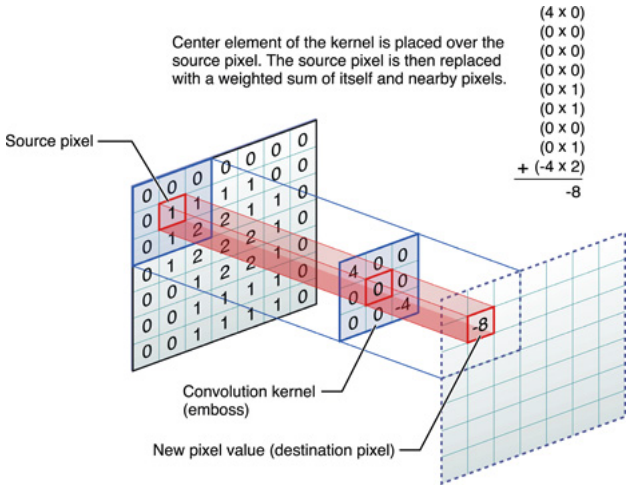


Figure 2.1: An example of 2D convolution, credits to vImage Programming Guide

$$
\begin{aligned}
R_1 &= I_1 * K_1 \qquad (2.1)\\
I_1 &= \begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \\
K_1 &= \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \\
R_1 &= I_1 * K_1 = \begin{bmatrix} 2 & 0 \\ 0 & 0 \end{bmatrix}
\end{aligned}
$$

$$
\begin{aligned}
R_2 &= I_2 \cdot K_2 \\
I_2 &= \begin{bmatrix} 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \\
K_2 &= \begin{bmatrix} 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix} \\
R_2 &= I_2 \cdot K_2 = \begin{bmatrix} 2 & 0 & 0 & 0 \end{bmatrix}
\end{aligned}
\tag{2.2}
$$

For example, this thesis hereby uses $*$ to represent the operation of convolution and $\cdot$ to represent the operation of matrix multiplication. In convolution, no preprocessing is needed for the input, therefore it could keep its spatial features during the operation. On the other hand, the input of matrix multiplication has to be flatten into a vector in order to validate the matrix multiplication. Equation 2.1 and 2.2 show that in order to detect the existence of diagonal (R_1, R_2) in a 3 by 3 image (I_1, I_2), the parameters for convolution (K_1) are 4 because the diagonal feature only needs to be detected in a 2 by 2 square. However, in order to attain the same result, the kernel of matrix multiplication (K_2) requires $4 \times 9 = 36$ parameters, with the input size of 9 and the output size of 4.

Usually, a CNN is built up with six types of layers according to some connection rules, and these six types of layer are: input, convolutional, Rectified Linear Unit (ReLU), pooling, fully connected and output. The Figure 2.2 demonstrates a standard Visual Geometry Group (VGG) model for image recognition, revealing that most of the computation happens in the convolutional layers. In a common CNN model, the convolutional layers are responsible for feature extraction and followed by ReLU layers in order to add non-linearity to the system. After several convolution layers and ReLU layers, a pooling layers will be interpolated into the model so that the system could reduce the dimensionality of the input and perform feature extraction on higher level. At last, the fully connected layers will take the results after all the convolution as input and gradually transform it into label-like inference. The convolutional layers are composed of many convolutional filters (kernels), each parameter in the filters is obtained after the training of backward propagation. Most of

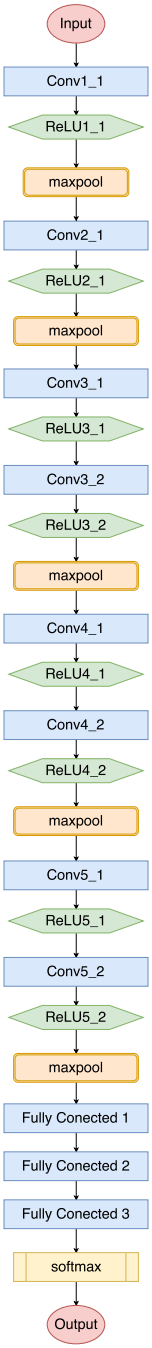


Figure 2.2: A standard VGG13 architecture. The post-fix number represents the layer number, for example, Conv3_1 means the first sublayer of the third convolutional layer.

the computation happens in the convolutional layers and the computational complexity is determined by the size of inputs and kernels. For instance, the 8-layer AlexNet[10] involves 60 million parameters and requires over 729 million Floating-point Operations Per Second (FLOPS) to classify one image per second. One of the problems this thesis tries to tackle is to reduce the unnecessary FLOPS based on some special conditions in convolution.

With the development of CNNs and Deep Learning, some new architectures have been developed for better accuracy on image recognition, and VGG[15] model proposed by University of Oxford is one of them. Opposite to the AlexNet, the size of convolutional filter in VGG becomes smaller and most of them are 3 by 3 filters. However, the contribution of VGG is exactly the smaller filters. By using multiple 3 by 3 convolutional filters, VGG can imitate the effect of the larger receptive field. VGG has been used in transfer learning for object localization tasks, making it a suitable starting point for vehicle localization in traffic applications, monitoring crowd movement for public security and so on. This is also the reason why this thesis selects several pretrained VGG models as the benchmark for different implementation. Accompanying the improvement of accuracy, the size of the model becomes larger, even type A of VGG model still contains 133 million parameters. That means heavy computational burden is not only for training the model but also for the implementation.

2.1.1 Sparse Convolutional Neural Network

There is a special representation of CNN named sparse Convolutional Neural Network (S-CNN). The sparsity of a CNN is the percentage of zero placeholder in its entire network and if the network has more than 50% sparsity, people can define it as an S-CNN. One of the useful features of S-CNN is: if a kernel is full of zeros, the result of the convolution on any input will also become zeros. In other words, with this predetermined characteristic, the simplification of S-CNN implementation becomes extracting non-zero values for meaningful calculation.

Even if the kernel is not totally filled up with zeros, the improvement of computational efficiency on S-CNN is also huge after some omission of calculation. For discrete, two-dimensional variables A and B, the following equation defines the convolution output C of A and B[20]:

$$C(j,k) = \sum_{p} \sum_{q} A(p,q)\, B(j-p+1, k-q+1) \tag{2.3}$$

where p and q run over all values that lead to valid subscripts of $A(p,q)$ and $B(j-p+1, k-q+1)$.

Based on this definition, it is possible to calculate the number of multiplications and additions during the convolution for a given input and kernel. If zero multiplications and additions could be removed from the convolution, the number of multiplications and additions required in a convolution could be defined as Equation 2.4 and 2.5:

$$N_1 = NZV \times NRF \tag{2.4}$$

$$N_2 = \begin{cases} (NZV - 1) \times NRF & NZV \geq 2 \\ 0 & NZV = 0, 1 \end{cases} \tag{2.5}$$

where N_1 and N_2 correspondingly represent the number of multiplications and additions required in the convolution, NZV represents the number of non-zero values in the kernel and NRF represents the number of receptive fields (convolutional areas).

According to Equation 2.4 and 2.5, the number of multiplication and addition is monotonically increasing with the number of non-zero values in the kernel. Table 2.1 shows the reduction of floating-point operations on sparsity. The number of receptive fields in this convolution is 9 and the number of non-zero values in the kernel is decreasing by one each time. This table hereby combines both multiplication and addition as the total computation and calculate the difference among different sparsity. As the percentage of computation reduction shown in the table, the computation decreases along with the increase of sparsity (less non-zero values in the kernel).

The generation of S-CNN comes from ReLU layer originally. The evolutionary neural network, which will be introduced in the next section, also yields the controllable sparsification during the training. The ReLU layer as the activation layer, intercepts the negative values of the input and turns them to be zeros. Because ReLU layers are added after every convolutional layer in VGG models, the latter layers could easily become highly sparse so that the sparsity of the model also increases.

2.1.2 Evolutionary Neural Network

From the above section, the potential of S-CNN to improve computational efficiency on convolution is obviously great. Then a new question may arise from this desire to simplify the convolution: how to make the weight tensors sparser. There are many ways to get this job done, and the method used by this thesis is evolutionary neural network[14]. Simply speaking, the evolutionary neural networks use iteration to update and optimize its architecture. The next generation of the network is modified based on the previous

Table 2.1: Convolution of a 3×3 kernel on a 5×5 input without padding, stride 1

Kernel Type	# of multiplication	# of addition	computation reduction
fully dense	81	72	0%
8 non-zero value	72	63	11.80%
7 non-zero values	63	54	23.50%
6 non-zero values	54	45	35.30%
5 non-zero values	45	36	47.10%
4 non-zero values	36	27	58.80%
3 non-zero values	27	18	70.60%
2 non-zero values	18	9	82.40%
1 non-zero value	9	0	94.10%
empty	0	0	100%

generation. If the accuracy of the previous generation of the network is above the accuracy requirement, the next generation of the network will be pruned or sparsified by some predefined rules; Otherwise, the next generation of the network will continue training to fill up the sparse tensors for higher accuracy.

The advantages of doing so is obvious: by imitating the evolution laws in biology, this Markov process will produce the best-optimized output under current constraints.

2.2 Hardware Platform

This section introduces the hardware platform to execute the CNN models, including normal GPU, CPU, ARM and Jetson TX1.

2.2.1 GPU

A Graphics Processing Unit (GPU) is a specialized electronic circuit that originally designed for high-speed graphics processing, including rapid memory reallocation and frequent image manipulation. The parallel structure of GPUs makes them more efficient when compared to general-purpose Central Processing Unit (CPU)s for algorithms processing large block of data. Therefore, pioneered by NVIDIA in 2007, GPU accelerators become an important part of supporting deep learning development. Figure 2.3 shows how GPU acceleration works for software application. In simple words, GPU will take

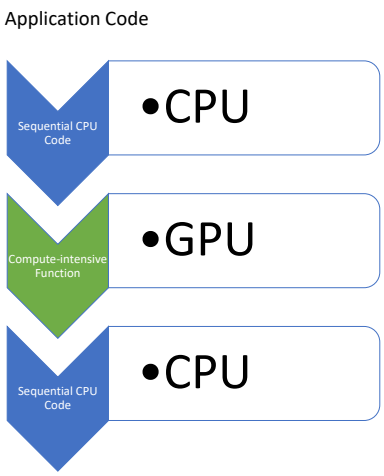


Figure 2.3: How GPU acceleration works(need to be replaced with a better one)

the computational tasks from CPU and perform operations independently. After that, the output of GPU will converge into main data stream for following process.

With the support of this acceleration, starting from 2010 by Dan Claudiu Cireşan at el.[4], implementing deep learning algorithms and CNN models on GPU is now the tendency, and this paper also shows the advantage of GPU on run-time in the following section. However, GPUs as the implementation platform also have their disadvantages. First of all, the price of high-speed GPU chip is not low enough for widely distributed application. The price of a normal GPU used for model training could be up to 2000 USD. Second, the power consumption of GPU is another an obstacle when people want to deploy them in some energy-insufficient environments. Based on the illustration of Figure 2.3, GPU cannot complete computational tasks alone without task assignment from CPU. That means a GPU computational system must include at least one CPU, therefore the power consumption of GPU system should be more than the general CPU system.

2.2.2 CPU

CPU has a longer history than GPU. In general, CPU represents a series of logical machine that can execute complicated computer programs. This definition could easily include the early stage computers which already exist before the widespread of the word "CPU", nonetheless, at least from 1960's, the word "CPU" has been widely used in electronic computer industry.

The performance of the CPU mainly depends on the the clock rate and the instructions per clock. It is worth mentioning that the clock rate cannot be increased dramatically because of the accompanying increased amount of heat dissipated by the CPU. Therefore, current improvement on computational ability of CPU is focusing on parallel computation, which includes instruction level parallelism and thread level parallelism. Meanwhile, the invention of Single Instruction Multiple Data (SIMD) and multi-core techniques also enormously strengthen the computational ability on chips.

Compared to GPU, CPU has a lot of advantages: higher accessibility, lower power consumption and price, and ability to work alone without adding control system. However, the parallel computation resource of CPU is much less than that of a normal GPU, which is a shortcoming in CPU-based system.

2.2.3 ARM

An Advanced RISC Machine (ARM) is an another type of micro processor and the hardware architecture of Reduced Instruction Set Computing (RISC) for computer processor. In other words, ARM can be treated as a weakened CPU which supports less instructions and owns less computational resource, but of course it has lower power consumption and price. These characteristics are desirable for light, portable and battery-powered devices. With the benefits mentioned above, the ARM processor has huge potential to become the suitable hardware solution to the topic of this thesis.

Figure 2.4 shows a prototype machine designed by Miovision Technologies which is using ARM as the computing core. The aim of this device is to collect and analyze visual traffic data with wide distribution, even in some energy-insufficient environments, such as mountain trails. As shown in the figure, this platform could be powered by rechargeable battery which validates it is used for low-power application. However, being limited by its processing power and memory, the ARM platform cannot afford a massive operating system running on it. Instead, to avoid extra overhead, this platform is running a nearly blank Linux OS, which only supports the execution of C++ programs after designated

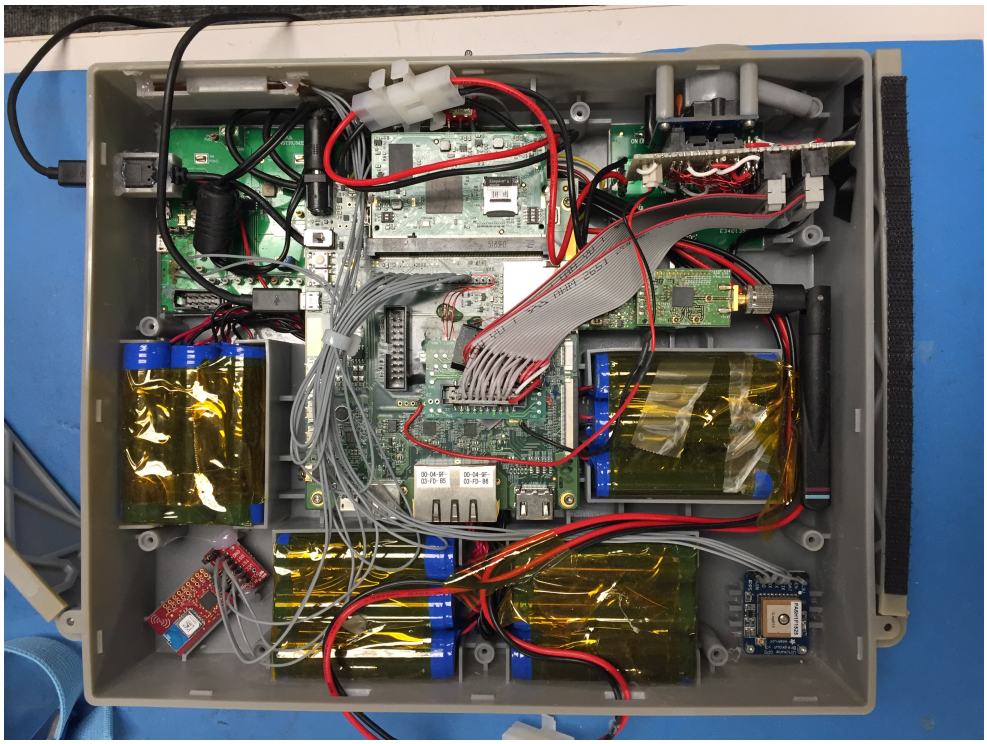


Figure 2.4: The ARM hardware platform used in the experiment provided by Miovision Technologies

Table 2.2: Module Technical Specification of Jetson TX1

GPU	NVIDIA Maxwell, 256 CUDA cores
CPU	Quad ARM A57/2 MB L2
Memory	4 GB 64 bit LPDDR4 25.6 GB/s
Data Storage	16 GB eMMC, SDIO, SATA
Other	UART, SPI, I2C, I2S, GPIOs
USB	USB 3.0 + USB 2.0
Connectivity	1 Gigabyte Ethernet, 802.11ac WLAN, Bluetooth

compilation. This is one of the disadvantages of ARM, which means it is not supported by mainstream deep learning frameworks and requires some dedicated native C++ implementations.

2.2.4 Jetson TX1

Generally speaking, Jetson TX1 is an embedded system instead of a specific kind of chip. Jetson TX1 is the latest AI computing module invented by NVIDIA. From Table 2.2, it is obvious that this module is the combination of GPU and ARM, in which the ARM processor takes charge of computational task assignment and the specific low-power GPU is in charge of real-time computing. This combination tactfully takes advantages of both GPU and ARM so that it can guarantee the run-time performance. Meanwhile, the choice of low-power GPU and ARM allows people to deploy their systems in some power-constrained environments.

Another point worth mentioning is that Jetson TX1 comprises larger memory and data storage, which allows it to support a more functional operation system. The benefits are obvious: with the help of functional operating system, the implementation could directly utilize the existing frameworks, accelerators, libraries with which people are already familiar. This advantage would greatly shorten the circle of development of applications and reduce the risk of infeasible solutions.

2.3 Software Platform

This section introduces two mainstream deep learning frameworks that will be used in the thesis: TensorFlow and Caffe.

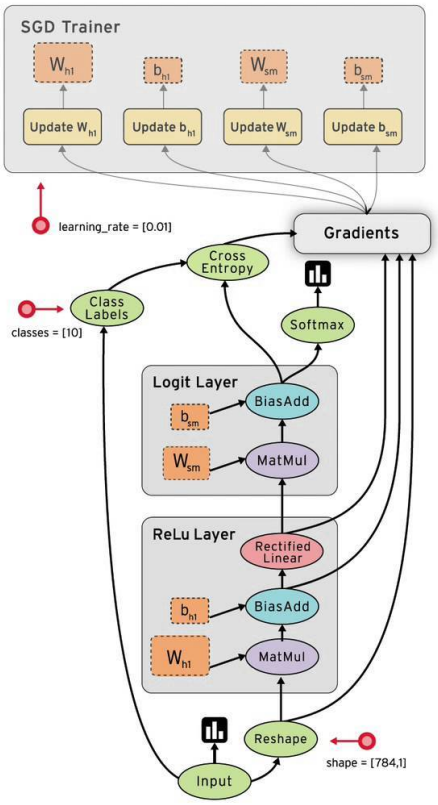


Figure 2.5: Demonstration of Nodes and Data Flow in Tensorflow

2.3.1 Tensorflow

TensorFlow[1][25] is an open source software library for numerical computation using data flow graphs. The most important concepts in TensorFlow are nodes and tensors. Tensors are the multidimensional data arrays that store the results after operations and will be treated as the input to next operation if applicable. Correspondingly, nodes in the graph represent mathematical operations according to the network structure.

Figure 2.5 shows an example of working paradigm of Tensorflow. The model in the figure is logistic regression model which is used for inference of an input with given size and trained by stochastic gradient descent method. The black lines with arrow represent the data flow in the model and different colorful blocks are different operations performed in the process, which are the nodes in Tensorflow's terminology.

As the most popular deep learning framework in the world, TensorFlow supports a great variety of models and owns a vigorous, growing community. Meanwhile, TensorFlow was originally developed by researchers and engineers working on the Google Brain Team, and those people are still maintaining TensorFlow and keep upgrading it. Based on this advantage, this thesis mainly uses TensorFlow as the benchmark to evaluate the runtime performance of the proposed solution.

2.3.2 Caffe

Caffe[8] is a deep learning framework developed by Berkeley AI Research and its community contributors. Caffe uses blobs[16] to store computational data, where a blob is an N-dimensional array stored in a C-contiguous fashion and can be synchronized between CPU and GPU. Besides, Caffe defines various layers[17] for different mathematical operations, including convolution, pooling, activation, loss, etc. Combining layers with blobs as their inputs/outputs forms the desired network.

One of the advantages of Caffe is its speed because it utilizes Basic Linear Algebra Subprograms (BLAS) to accelerate the inner matrix computation. However, the selected ARM platform for experiment does not support this Application Programming Interface (API) standard, so the Caffe library cannot be transplanted to the ARM platform directly. This thesis uses Caffe with Jetson TX1 to achieve better runtime performance because NVIDIA has already developed an acceleration library named TensorRT on Jetson TX1 for caffemodel.

Chapter 3

Problem Formulation

In order to design a deployable traffic monitor system, there are two aspects that deserve researchers' attention. First, To achieve wide distribution, it means the systems could be deployed at the locations that lack infrastructure. Therefore, the system should be equipped with self-powering and power-harvesting components, and at the same time, the power of the whole system should be as low as possible. Second, the expected system should have reliable runtime performance, in other words, its computational speed should be fast enough to support dynamic image recognition at a minimum frame rate with the acceptable accuracy. However, the above two points are relevant. The higher computational speed requires more computational units, which would increase the power of the system and vice versa. Therefore, how to balance these two requirements is the main problem of the research in this thesis. This chapter introduces what specifications of the expected system are, including the power consumption and runtime performance requirements.

3.1 Power Consumption

Since users hope that the system could be deployed everywhere and maintain self-operation for more than a year (battery wastage neglected), the maximum power of the system should depends on how much power a modern power harvest technique could provide. According to the weather data in Waterloo region[21], this thesis drew Figure 3.1 and 3.2 for further analysis.

Even though the monthly wind speed is high during winter based on Figure 3.1, when applying small-scale turbines to wind power harvesting, the total energy provided by wind

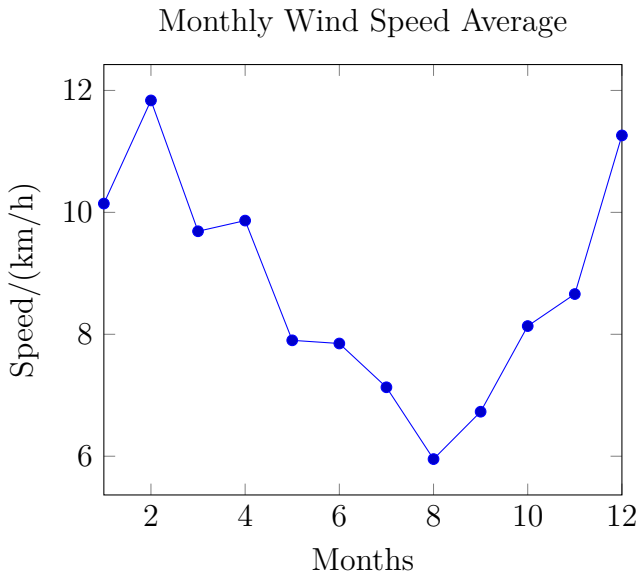


Figure 3.1: Monthly Wind Speed Average in 2016 (Waterloo Region)

power in 2016 is less than 4kWh, which means the utilization of wind power could support the device with power consumption below 0.5W.

On the other hand, based on the result of Figure 3.2, even applying small-scale solar panel to solar power harvesting, the total energy provided by solar power in 2016 could be up to 47kWh. Thus the device powered by solar panel could also have the power consumption more than 5W.

Based on the calculation above, both wind power and solar power cannot provide much energy to the system. This fact also validates that the expected system should be low-power enough to become widely deployable. Since it is still an exploration stage of the research now, this thesis will adopt the reasonable rough estimation of 6W (combining wind power and solar power, then rounding up to an integer) as the ideal power consumption of the expected system.

3.2 Runtime

According to the related regulations governing traffic and transportation, such as National Electrical Manufacturers Association (NEMA)[2], the traffic optimization system should

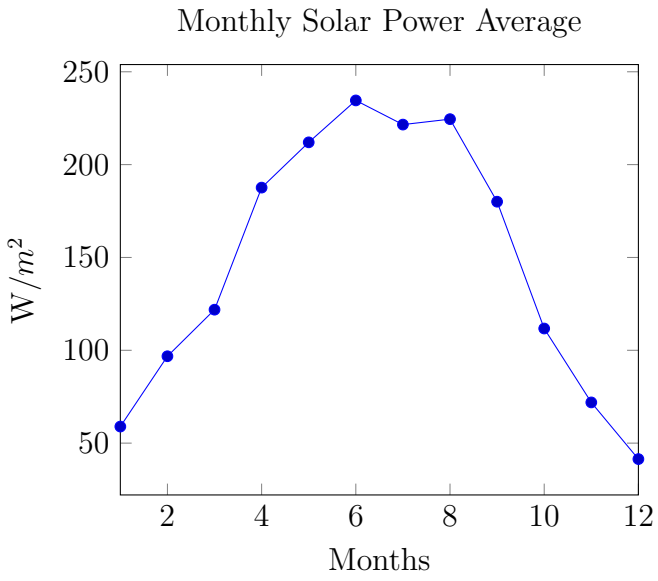


Figure 3.2: Monthly Solar Power Average in 2016 (Waterloo Region)

have the ability to process at least 5 frames per second. In real world, the frame rate is acceptable for traffic signal processing and decision making, therefore, 200ms per frame is used as the benchmark for runtime requirement in the following sections.

Chapter 4

Solution Strategy and Experiment Setup

This chapter will introduce mechanism of optimization on S-CNN and general CNN, and also setup the environment required for the following experiments.

4.1 Optimization on S-CNN

It has been mentioned in Chapter 2 that S-CNN contains tremendous potential to be optimized, it is worth research how people can optimize the computation on an S-CNN.

4.1.1 Motivation

Since many well-optimized deep learning frameworks exist today, it is needed to know whether current frameworks have been optimized for S-CNN. After checking the related documents of Tensorflow, it has been found that Tensorflow does not get S-CNN computation optimized to a system level. Although Tensorflow provides sparse matrix operations as its built-in functions, the implementation process will not automatically call those functions. Besides, considering the limitation of ARM platform, current frameworks are established with many dependencies on normal desktop Ubuntu system and thus become so difficult to be transplanted to a tiny, less functional operating system. Hence, to further

Figure 4.1: The process of S-CNN optimization from pre-trained model to deployable code

reduce computation and become adjusted to ARM platform, a native C++ solution supported by Standard Template Library (STL) is the main optimization method proposed in this thesis.

4.1.2 Details

Figure 4.1 displays the process of S-CNN optimization during the proposed implementation. First of all, the pre-trained model as the input will be analyzed by a preprocessor. The responsibility of the preprocessor is to select the useful information and send it to a code-generator in a compact data format. At last the code-generator will generate the optimized code for specific implementation environment.

The preprocessor traverses the S-CNN, detects the non-zero values, extracts the non-zero values and their position information, then reorganizes the kernel data format for the code-generator in the following steps. Figure 4.2 shows the indexing of a $3 \times 3 \times 3$ kernel, which demonstrates how this thesis represents the values of each position. To generate a sparse representation of the data, the transformation schema must contain both a position and a value. The position index can be obtained by unwrapping the 3d convolutional kernel into three separate matrices and these indices are going to be used in sparse formatting.

An example of sparse formatting is shown in Figure 4.3. After analysis on the sparsity

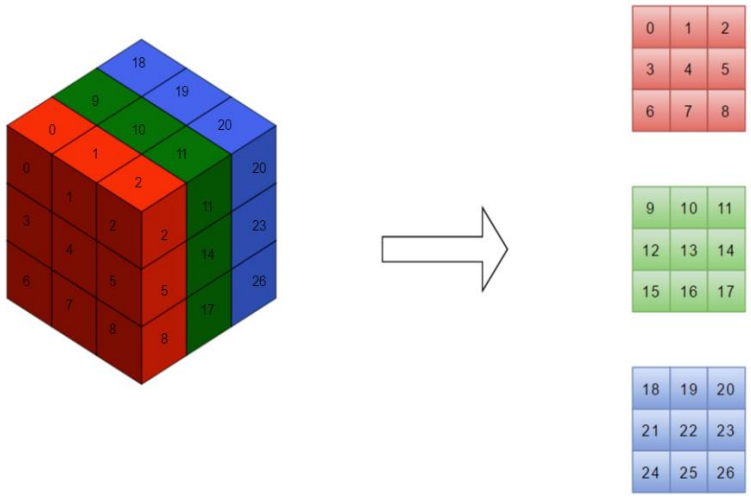


Figure 4.2: Indexing of A 3 by 3 Input

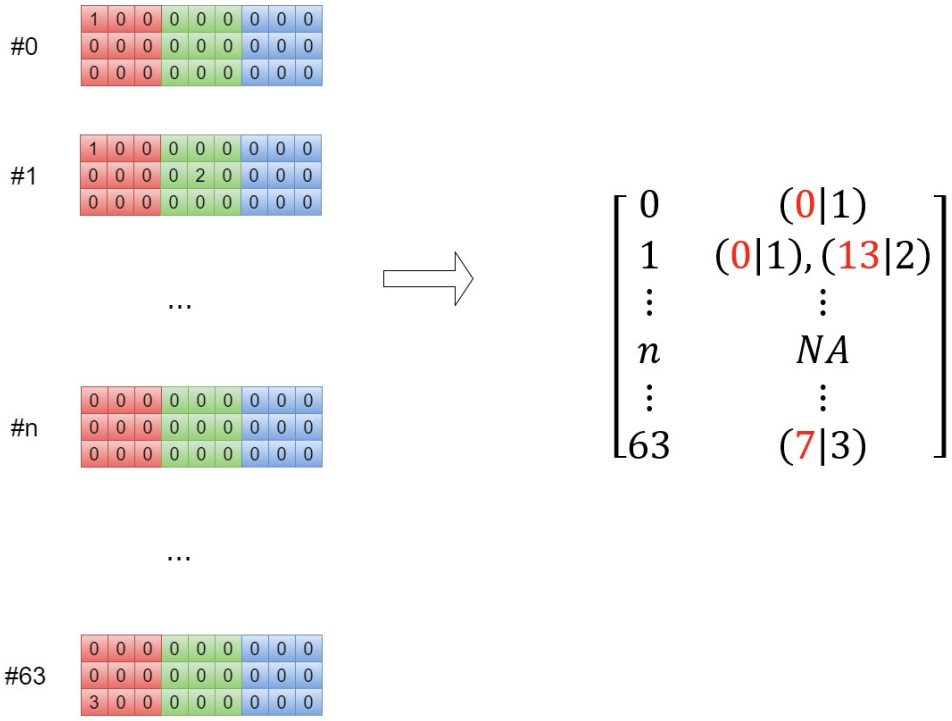


Figure 4.3: An Example of Sparse Transformation

of the kernel, a new data format is generated by combining the non-zero values and their position in the kernel. Kernel #0 is represented as (0|1) because there is a non-zero value in position 0 (where the red digit represents the position in kernel). NA is used to indicate that all entries are zero for the kernel.

Using the sparse representation illustrated in Figure 4.3, the code-generator can easily avoid unnecessary calculations. For example, once the code-generator knows that any kernels in the convolutional layer are empty, it can just skip all the computation for those output channels leaving the result all zeroes. Moreover, with further analysis, this output channel will become a part of the input for the next convolutional layer, which means even if the kernels of next layer are not all zero for this input channel, the related computation is also avoidable and could be set to zero during processing. Figure 4.4 is used for example to illustrate the generation process. This figure displays how to cache the m $\times$ n $\times$ 3 image into an array according to the convolutional area. Then combine with the sparse representation, extract the non-zero value for arithmetic operation. Finally, auto-generated code is attained.

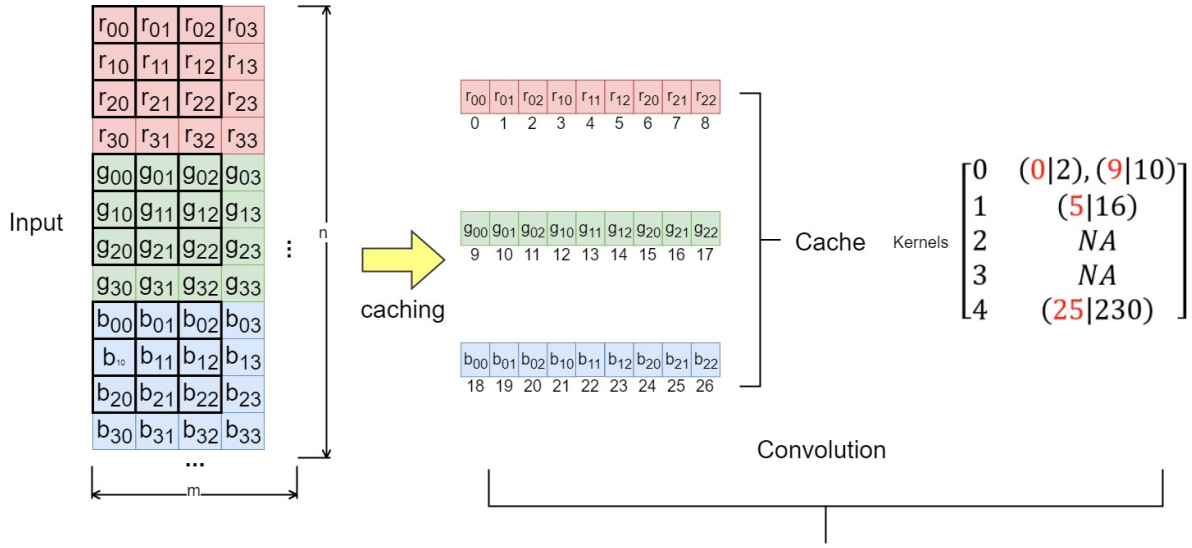
Moreover, the proposed solution uses some caching techniques to reduce memory consumption and improve computational efficiency. The idea of caching is based on memory addressing. Addressing in a massive block of memory takes more time than that in a smaller block of memory. Therefore, the proposed solution copies the values in the convolutional area into a smaller array, turning the wide-range addressing to the smaller one.

4.2 Optimization on General CNN

During the research of the optimization on S-CNN, the CV team in Miovision also wants to know the performance of network pruning besides sparsification. Thus, the proposed optimization solution should also include optimization on a general CNN.

4.2.1 Motivation

Because current frameworks have many active communities, they have been optimized well enough for general CNN on the normal platform. However, as mentioned in the previous section, the main obstacle is portability. Besides, transplantation is also complex and time-consuming because all their dependencies have to be recompiled for the specific platform hierarchically. To play a better supporting role with S-CNN optimization, a native C++ optimization solution is proposed for general usage.




```
...
Output[0][i] = Cache[0] * 2 + Cache[9] * 10;
Output[1][i] = Cache[5] * 16;
//Output[2] and Output[3] are all set to zeroes because kernel_2 and kernel_3 are all zeroes
Output[4][i] = Cache[25] * 230;
...
```

Figure 4.4: Code Generation Process

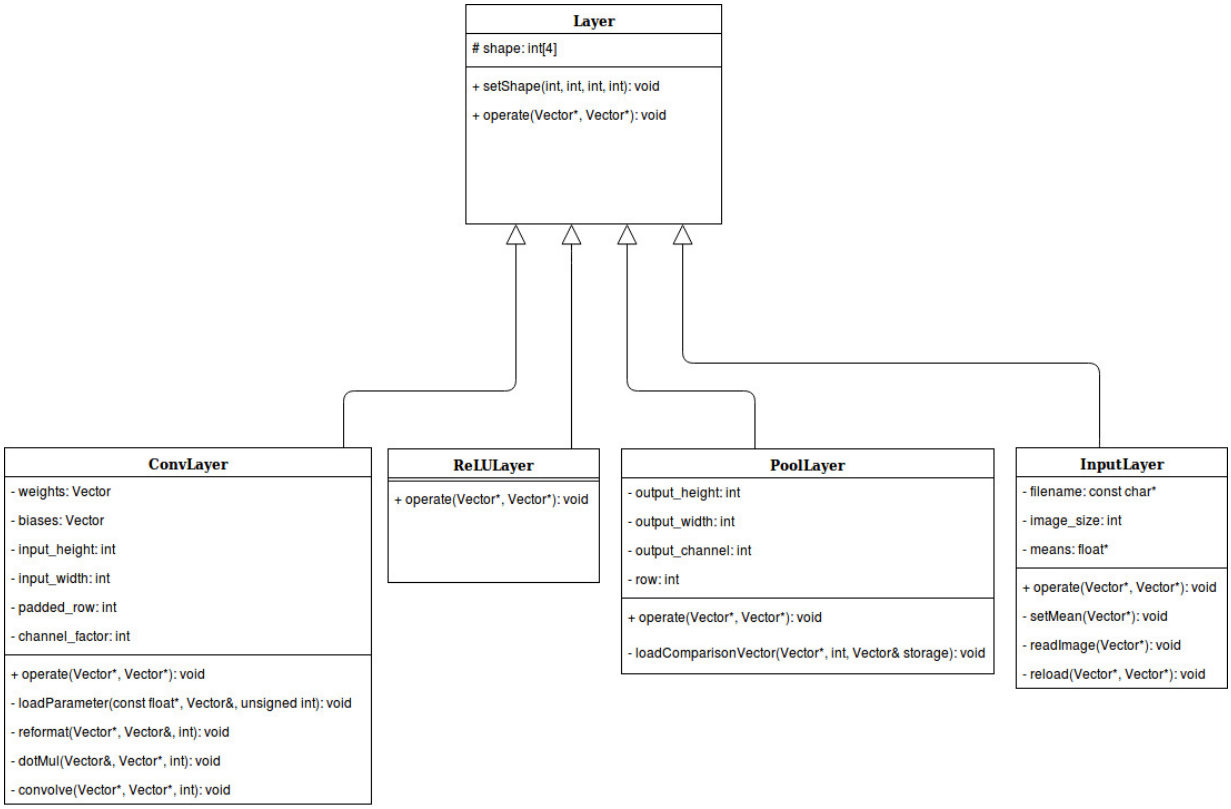


Figure 4.5: Class Diagram of General Optimization

4.2.2 Details

The main idea of general optimization is constructing a C++ implementation by simulating current deep learning frameworks. Rather than optimizing training and implementation both at the same time, the author's idea is trying to reduce the size and procedure of the executable file which only aims at implementation.

In effect, the proposed general optimization is to utilize the hierarchical architecture of Caffe but remove the unnecessary or unaccessible components of it. Given a specific model, it is possible to slightly optimize the run-time consumption after analysis. For example, if the network is sequentially executed, it is possible to reduce the number of storage blobs by clearing previous input blob for the output of next layer instead of assigning a storage blob for each layer. Figure 4.5 displays the class design of general optimization.

Eventually, by combining both S-CNN optimization and general optimization, this

Table 4.1: Model Specification

Model	S-CNN1	S-CNN2	Reduced CNN
Type	VGG	VGG	VGG
Foreground Accuracy	96%	94%	92%
Sparsity	94%	98%	0%
Number of Parameters	14714688	14714688	60315

thesis concludes a work schema as shown in Figure 4.6.

4.3 Experimental Environment

This section will explain what kinds of models are going to be used in the following experiments, and introduce the software-hardware environment setup.

4.3.1 Model

In the following experiments, there are three different CNN models in use: S-CNN with 94% sparsity, S-CNN with 98% sparsity and reduced CNN (pruning instead of sparsification). All of them are VGG models with foreground accuracy above 92% which has ensured the accuracy for image recognition application, and only conv1-1 to conv5-3 layers are selected for experiments to the effect of a series of optimization aimed at CNN. Table 4.1 shows the specification of those models. Being limited by the time, this thesis could only conduct the experiments on two different sparse models. Models with more various sparsity are still training and the same experiments will be conducted on them in the future.

4.3.2 Software Platform

In the following experiments, the author used Tensorflow for benchmark and Caffe to support Jetson TX1 implementation. Meanwhile, the proposed sparse optimization utilized Eigen[22], which is a C++ template library for linear algebra, as the support to S-CNN optimization. Table 4.2 lists the information of the software.

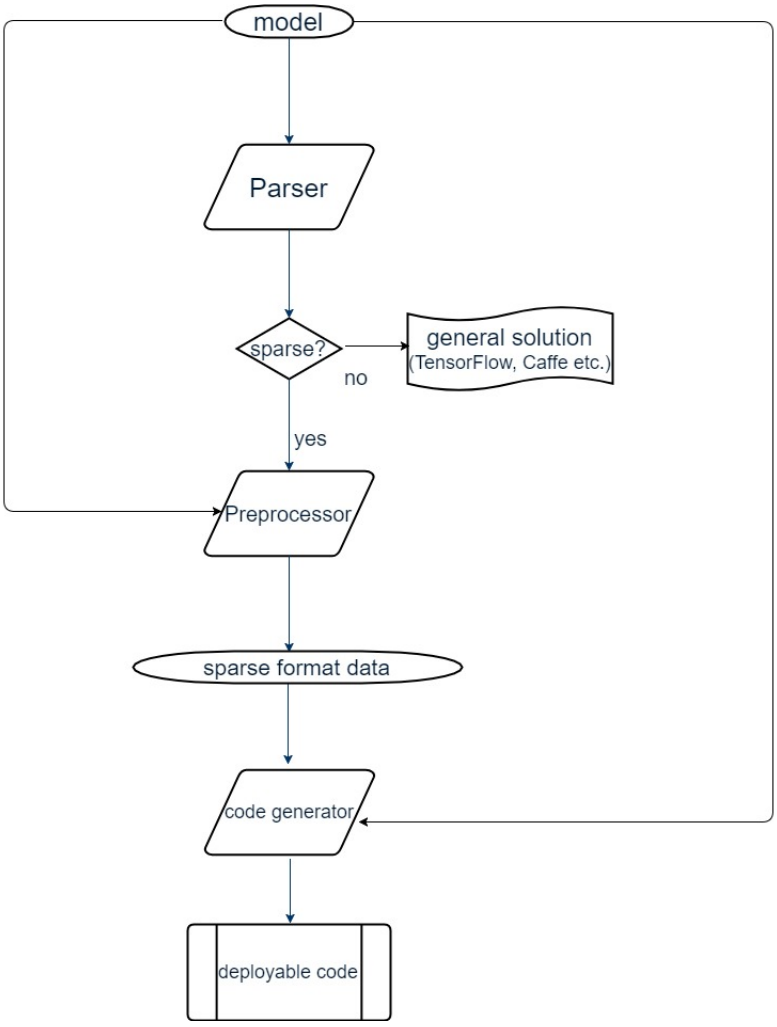


Figure 4.6: Proposed Work Schema

Table 4.2: Software Specification

Software	Tensorflow	Caffe	Eigen
Version	rc1.0	1.0	3.3
API	Python	C++	C++

Table 4.3: Hardware Platform Spec

System	AWS Server	Desktop	ARM Platfrom	Rudi Embedded System
Chip	TITAN X	Intel i7-4770	Freescale MCIMX6UL-EVK	Jetson TX1
RAM	12GB	16GB	512MB	4GB
Cost (core chip)	1200USD	312USD	∼200CAD	299USD
Clock Speed	1417MHz	3.4GHz	696MHz	N/A
Storage	N/A	1TB	4GB (via uSD Card)	16GB

4.3.3 Hardware Platform

In the following experiments, four different hardware platforms are used and the run-time performance of them are observed. Both GPU and CPU platform are power-consuming and not satisfied with the design requirement, therefore they are used for benchmark and comparison. The ARM platform is designed by Swift Lab, which aims at low-power environment. Besides, the Jetson TX1 has been integrated in an embedded system designed by Connecttech Technologies for the following experiments. Table 4.3 shows the specification of each platform.

Chapter 5

Experimental Results

In this chapter, five experiments are introduced and explained. They are GPU and CPU benchmark, effectiveness of S-CNN optimization, the feasibility of ARM platform, the feasibility of Jetson TX1, and power measurement.

5.1 GPU and CPU Platform Benchmark

This experiment is to set up the benchmark of GPU and CPU for the following experiments. The reason of setting GPU and CPU as benchmark is because their power is much higher than the expectation. However, they are the most popular deep learning hardware platform today, which makes them valuable as the benchmark.

Because of the excellent computational ability of GPU, the GPU platform should be able to decrease the runtime by at least one order of magnitude compared to the runtime of CPU platform. The CPU implementation in multi-threading mode should also be faster than that in single thread mode. At the same time, the runtime of implementation on both GPU and CPU platform should depends on the number of parameters in the models.

The results of experiment are shown in Table 5.1 and Figure 5.1. It is obvious that the runtime of a given model on a given hardware platform is related to the size of the model and the computational ability of that platform. For example, as shown in both Table 5.1 and Figure 5.1, S-CNN2 and S-CNN1 share similar runtime on GPU platform (13ms vs. 14ms). However, the performance of Reduced CNN is much better and only requires one third of the runtime of sparse models. When compared the number of parameters of them in Table 4.1, Reduced CNN is much smaller than those sparse models and this fact

Table 5.1: GPU and CPU Benchmark

Hardware	Software	Model	Runtime/(ms)
GPU	Tensorflow	S-CNN2, 98%	13
		S-CNN1, 94%	14
		Reduced CNN	4
CPU, 7 threads	Tensorflow	S-CNN2, 98%	706
		S-CNN1, 94%	713
		Reduced CNN	81
CPU, 1 thread	Tensorflow	S-CNN2, 98%	2370
		S-CNN1, 94%	2374
		Reduced CNN	227

indicates that the size of model will impact the runtime performance. Being limited by the computational ability, even the CPU platform with multi-threading could not beat the GPU platform on runtime.

5.2 Effectiveness of S-CNN Optimization

This experiment is designed to verify the effectiveness of CPU-based S-CNN optimization. Once the CPU-based S-CNN optimization has been proven as effective, the process of experiment will be moved forward to feasibility of implementation on ARM platform.

Because the CPU-based S-CNN optimization is well-optimized for S-CNN on the architecture level, it should have better performance than CPU-mode Tensorflow. Meanwhile, the runtime of this optimization algorithm should also be decreased with multi-threading.

The results of experiments are shown in Table 5.2 and Figure 5.2. As indicated in Table 5.2, the proposed optimization algorithm outperforms Tensorflow on both two sparse models. the speed of the proposed algorithm is almost as twice fast as that of Tensorflow on model with 94% sparsity and 5 times faster on model with 98% sparsity. Compared to the non-optimization of Tensorflow on sparse models, the result of this experiment verified the effectiveness of the proposed optimization on highly sparse models. However, Figure 5.2 tells the result that the optimization algorithm is still not good enough because the runtime of such an algorithm would greatly increase wiht slight drop of sparsity (from 118ms to 490ms, when sparsity drops from 98% to 94%).

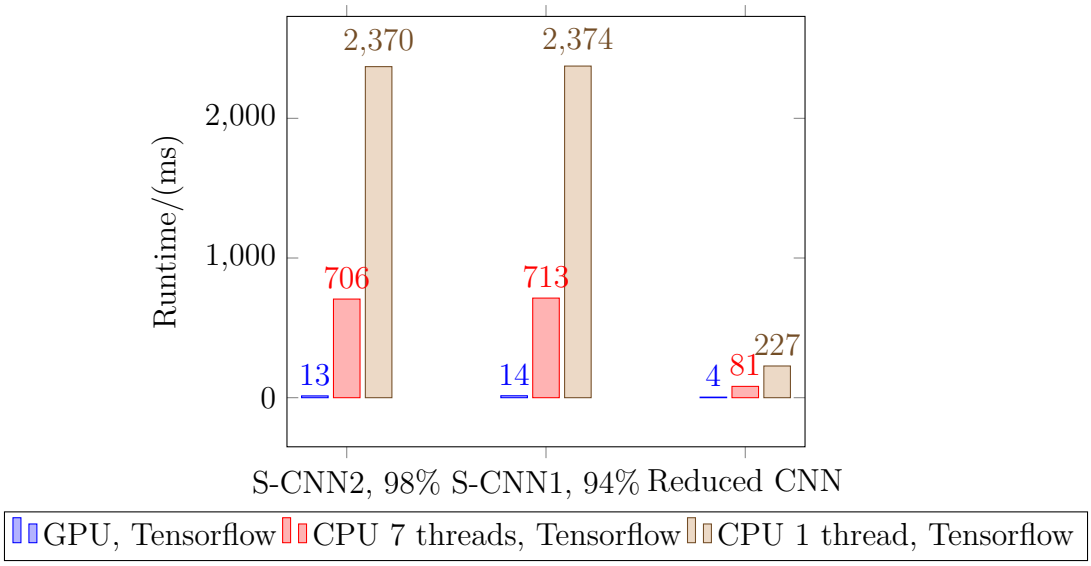


Figure 5.1: GPU and CPU Benchmark

5.3 Feasibility of ARM Platform

This experiment is designed to verify the feasibility of CNN real-time implementation on ARM platform. Being limited by the operating system on the platform, transplanting mainstream deep learning framework onto the ARM platform is quite difficult. On the other hand, since the S-CNN optimization algorithm has been proven that it outperforms Tensorflow on sparse models, the implementation on ARM platform will adopt the proposed native C++ solution.

Even though the S-CNN has greatly improve the computation, the limitation of hardware itself may bring bad impact over the optimization on algorithm. The runtime of ARM platform may be out of the expectation.

The results of experiment are shown in the Table 5.3 and the Figure 5.3. The results of experiment have verified the hypothesis that the selected ARM platform could not achieve the goal of runtime for real-time application. The processing speed of this ARM platform is quite slow (more than 40 second to process a frame) because it has lowest clock frequency among these four hardware platform. Except for low clock frequency, the ARM platform is also constrained by small RAM and simple instruction set, which makes it impossible to process massive computation in a short instructional period. On the other hand, the code immigration from CPU platform to ARM platform is successful, and the proposed S-CNN

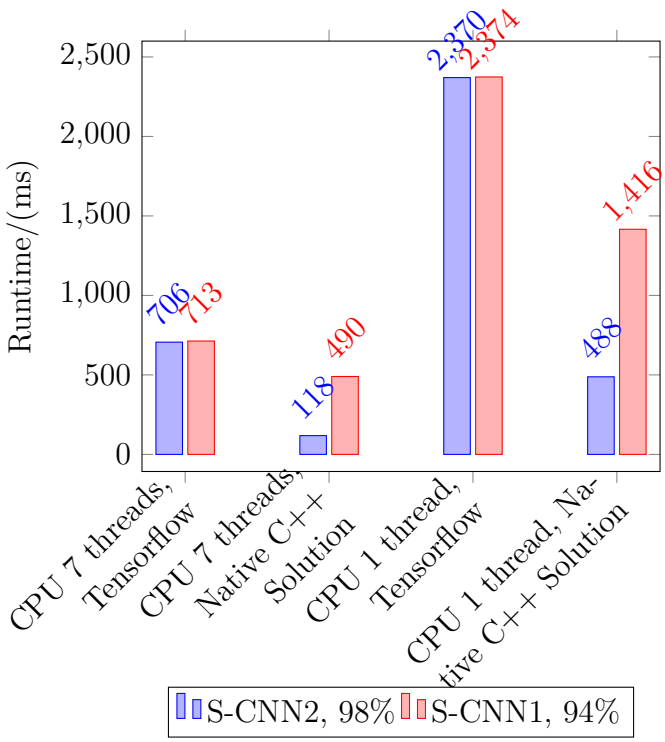


Figure 5.2: Effectiveness of S-CNN Optimization

Table 5.2: Verification of S-CNN Optimization

Hardware	Software	Model	Runtime/(ms)
CPU, 7 threads	Tensorflow	S-CNN2, 98%	706
		S-CNN1, 94%	713
	Native C++ Solution	S-CNN2, 98%	118
		S-CNN1, 94%	490
CPU, 1 thread	Tensorflow	S-CNN2, 98%	2370
		S-CNN1, 94%	2374
	Native C++ Solution	S-CNN2, 98%	488
		S-CNN1, 94%	1416

Table 5.3: Feasibility of ARM Platform

Hardware	Software	Model	Runtime/(ms)
CPU, 1 thread	Tensorflow	S-CNN2, 98%	2370
		S-CNN1, 94%	2374
		Reduced CNN	227
	Native C++ Solution	S-CNN2, 98%	488
		S-CNN1, 94%	1416
		Reduced CNN	1548
ARM	Native C++ Solution	S-CNN2, 98%	43703
		S-CNN1, 94%	175792
		Reduced CNN	40981

optimization algorithm keeps working well on sparser model.

5.4 Feasibility of Jetson TX1 Platform

This experiment is designed to test the feasibility of real-time CNN implementation on Jetson TX1 platform. Because this hardware platform is released to public recently, the author do not have enough time to implement many models on this platform. On the other hand, NVIDIA released a related acceleration library TensorRT[23] for pre-trained CNN implementation, which could cooperate with Caffe to obtain excellent runtime performance. According to the result of GPU and CPU benchmark experiment, current mainstream deep learning frameworks work well with small model and have less runtime than the proposed general CNN optimization. Thus, the combination of Caffe and reduced CNN is inferred

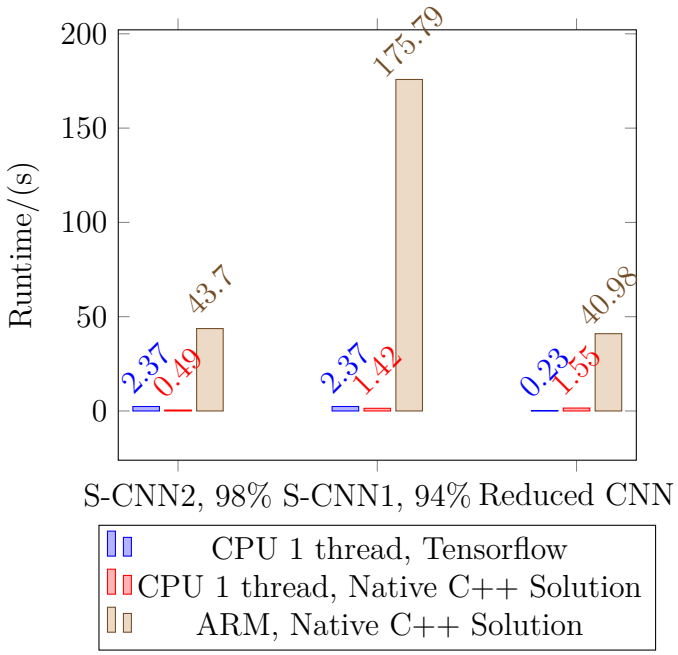


Figure 5.3: Feasibility of ARM Platform

Table 5.4: Feasibility of Jetson TX1 Platform

Hardware	GPU	CPU, 7 threads	ARM	Jetson TX1
Software	Tensorflow	Tensorflow	Native C++ Solution	Caffe + TensorRT
Model	Reduced CNN	Reduced CNN	Reduced CNN	Reduced CNN
Runtime/(ms)	4	81	40981	12

as the best combination on Jetson TX1 based on current knowledge.

Because Jetson TX1 is the combination of GPU and ARM, its runtime should be guaranteed by the computational ability of its embedded GPU. However, the GPU of Jetson TX1 platform is not as powerful as the selected GPU platform in this thesis by comparing the number of their computational cores (256 CUDA cores vs. 3584 CUDA cores). Thus the runtime of Jetson TX1 platform should be more than that of normal GPU platform.

As the result shown in Table 5.4, the Jetson TX1 platform has extraordinary performance (12ms per frame) compared to the best case of CPU platform (81ms per frame). However, the processing speed of the Jetson TX1 platform is still twice slower than that of a normal GPU platform, which reflects that the computational resource, such as the number of parallel computational cores, has great impact on runtime.

5.5 Power Measurement

This experiment is designed for measuring the power consumption of each platform. According to the specification of the data sheet and the real data from measurement, the results could be used as a reference for future design evaluation.

According to the specification of NVIDIA TITAN X and Intel i7-4770, their power consumption is out of the design requirement, so that the power of GPU and CPU platform is not measured and assumed the same as mentioned in the specification sheet. The ARM platform designed by Swift Lab for low-power environment should have the best power performance among these four platforms.

The results of experiment are shown in the Table 5.5 and Figure 5.4. On one hand, the VIP lab at the University of Waterloo has conducted a simple experiment for power measurement of Jetson TX1 by recording the data from related monitoring software. Those data reveal that the power consumption of Jetson TX1 platform varies from 7 watts to

Table 5.5: Power Measurement

Hardware	GPU	CPU	ARM	Jetson TX1
Power/(W)	250[19]	35[18]	1.4	15

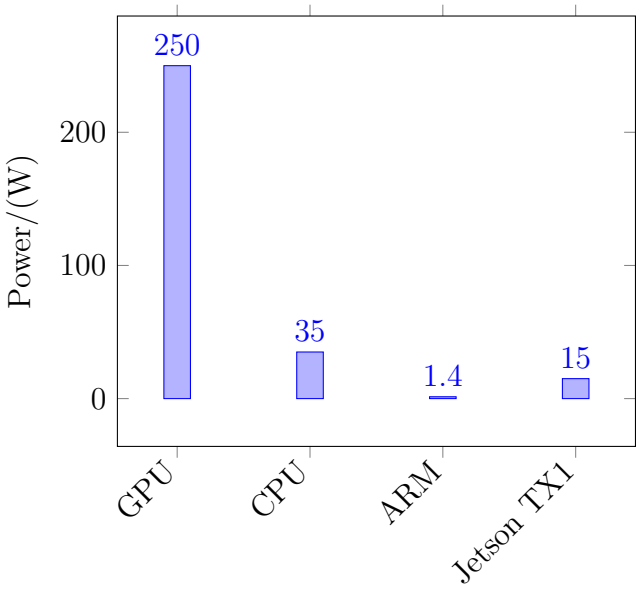


Figure 5.4: Power Measurement

14 watts. Therefore, to retain more design margin, this thesis adopted 15 watts as the standard power consumption of Jetson TX1 platform.

On the other hand, the ARM platform designed by Swift Lab fulfills the power requirement proposed in Chapter 3. As displayed in Figure 5.5, this platform has an active status with around 1.4 watts power consumption. Even during the communication session of this platform, its power consumption has just increased from 1.4 watts to 1.8 watts. It is obvious that this device has excellent low-power performance while sacrifices computational speed. Therefore, the improvement of this platform should be focused on increasing the power for more computational resource.

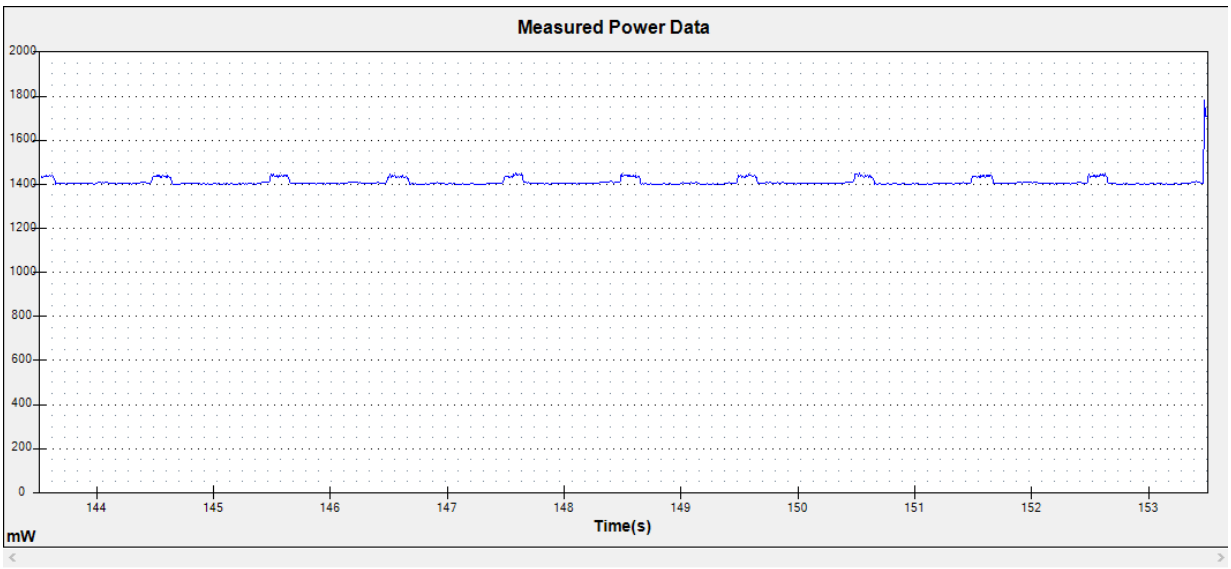


Figure 5.5: Power Consumption of ARM Platform during Runtime

Chapter 6

Conclusion and Discussion

6.1 Hardware Platform for Research

According to the experiments in the previous chapter, the most suitable hardware platform for deep learning research is still the normal GPU. Benefited from the multiple cores and parallel computation, the GPUs could achieve astonishing computational speed on 2-D convolution.

In research environment, computational speed is the main factor in choosing hardware platform. Meanwhile, because in most cases the research environment could provide stable and sufficient power for high-speed computation, the power consumption would not become the constraint for the platform selection. To accelerate the training process, the GPU platform is recommended for researchers.

6.2 Hardware Platform for Low-power Control

Unlike the GPU platform, the ARM platform mentioned in the previous chapters requires very low power to complete the computational tasks. However, it needs much longer processing time.

In effect, the ARM processor should take the place of CPU to become the main processor in the smart traffic applications. That is what Jetson TX1 did in its architecture. In Jetson TX1, the low-power ARM takes charge of computational task assignment and the low-power GPU is focusing on the massive computation.

6.3 Hardware Platform for Widely Distributed Vehicle Detection System

Based on the experiments so far, the Jetson TX1 Platform has the greatest potential to become the applicable hardware platform for the proposed widely distributed system in this thesis. Even though its power consumption is a little higher than the requirement, this does not impede the hardware architecture becoming the effective one for real-time CNN implementation.

At least in the near future, the power of the Jetson TX series module could become low-power enough to complete such amount of computation in the acceptable time. The recent release of Jetson TX2 added the functionality of clock speed adjustment, which allowed users to adjust the clock speed for faster computation or lower power consumption.

6.4 Software Improvement

Another gain from the previous experiments is that the current deep learning frameworks have not pay much attention to the potential of S-CNN for reducing the computation. Because the model is fixed during the implementation, and it is allowed to generate well optimized code after analysis, current frameworks should take the S-CNN scenario into consideration as well.

Although the experiments show the Tensorflow works well with the pruning network, which is fully dense, it does not mean Tensorflow with the reduced network could always become the substitution for S-CNN implementation. It should be noticed that the accuracy of pruning network has decreased compared to those S-CNNs. The "0" placeholders in the S-CNNs are not meaningless, instead they contain the necessary information for object detection and classification. In addition, the larger architecture of S-CNNs also helps improve the accuracy. Therefore, in some specific cases with requirement of higher accuracy, the pruning network may not be the best option. That is the reason why preprocessing the model and the related analysis is recommended. With better understanding of the model, the designers could choose the most suitable implementation methods for their application.

Chapter 7

Future Work

Though the author has verified many combinations of hardware, software and models, and also drawn some conclusions based on current technique, there is some future work that needs to be done.

7.1 Research on Jetson TX Series

Since the Conclusion Chapter has mentioned that the Jetson TX series has great potential to implement real-time CNN computation, such hardware platform deserves more future attention. Actually, the matched library, TensorRT is also evolving, which has already released 2.0 version for developers. One of the reasons why the research could not conduct more experiments on Jetson TX1 is because the TensorRT still has many bugs and the source code has not become open source yet. This deep learning system invented by NVIDIA is worth the wait.

7.2 Dedicated Convolutional Chip

Another idea of hardware acceleration for CNN implementation is the invention of a dedicated computational chip based on convolution structure. The basic idea is to simulate the convolution process physically. By applying a large number of multiplication unit to simulate the convolutional kernel, the data could be piped in with higher efficiency. The

simulation could be done on FPGA[6] to see the potential impact on CNN implementation. In effect, more and more companies has aimed at the next generation of deep learning hardware platform. For example, Tensor Processing Unit (TPU)[9] designed by Google is a shock to current situation leading by NVIDIA and its GPUs. The author hopes to get access to more hardware platforms and understand their performance during implementation.

7.3 S-CNN Optimization on GPU

Someone may have noticed that the proposed S-CNN optimization is based on CPU. The author also believes that this optimization algorithm could be applied to GPU acceleration. Future improvement of this optimization method should include the GPU implementation and multi-threading adjustment.

References

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. Tensorflow: A system for large-scale machine learning. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI). Savannah, Georgia, USA*, 2016.

[2] National Electrical Manufacturers Association. *NEMA Standards Publication TS 2-2003 (R2008)*. National Electrical Manufacturers Association, 2008.

[3] Andrew S Cassidy, Paul Merolla, John V Arthur, Steve K Esser, Bryan Jackson, Rodrigo Alvarez-Icaza, Pallab Datta, Jun Sawada, Theodore M Wong, Vitaly Feldman, et al. Cognitive computing building block: A versatile and efficient digital neuron model for neurosynaptic cores. In *Neural Networks (IJCNN), The 2013 International Joint Conference on*, pages 1–10. IEEE, 2013.

[4] Dan Claudiu Cireşan, Ueli Meier, Luca Maria Gambardella, and Jürgen Schmidhuber. Deep, big, simple neural nets for handwritten digit recognition. *Neural computation*, 22(12):3207–3220, 2010.

[5] Vincent Dumoulin and Francesco Visin. A guide to convolution arithmetic for deep learning. *arXiv preprint arXiv:1603.07285*, 2016.

[6] Pedro Ferreira, Pedro Ribeiro, Ana Antunes, and Fernando Dias. Artificial neural networks processor–a hardware implementation using a fpga. *Field Programmable Logic and Application*, pages 1084–1086, 2004.

[7] Paul Fieguth. *Statistical image processing and multidimensional modeling*. Springer Science & Business Media, 2010.

[8] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. Caffe: Convolutional architecture for fast feature embedding. *arXiv preprint arXiv:1408.5093*, 2014.

[9] Norman P Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, et al. In-datacenter performance analysis of a tensor processing unit. *arXiv preprint arXiv:1704.04760*, 2017.

[10] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[11] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *Nature*, 521(7553):436–444, 2015.

[12] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[13] Giffinger Rudolf, Christian Fertner, Hans Kramar, Robert Kalasek, Natasa Pichler-Milanovic, and Evert Meijers. Smart cities-ranking of european medium-sized cities. *Rapport technique, Vienna Centre of Regional Science*, 2007.

[14] Mohammad Javad Shafiee and Alexander Wong. Evolutionary synthesis of deep neural networks via synaptic cluster-driven genetic encoding. *arXiv preprint arXiv:1609.01360*, 2016.

[15] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[16] Website. Caffe blob introduction. http://caffe.berkeleyvision.org/tutorial/net_layer_blob.html, 2014. Accessed June 6, 2017.

[17] Website. Caffe layer catalogue. http://caffe.berkeleyvision.org/tutorial/layers.html, 2014. Accessed June 6, 2017.

[18] Website. Specification of i7-4770, intel. http://ark.intel.com/products/75122/Intel-Core-i7-4770-Processor-8M-Cache-up-to-3_90-GHz, 2016. Accessed July 26, 2017.

[19] Website. Specification of titan x, nvidia. https://www.nvidia.com/en-us/geforce/products/10series/titan-x-pascal/, 2016. Accessed July 26, 2017.

[20] Website. 2-d convolution. https://www.mathworks.com/help/matlab/ref/conv2.html?requestedDomain=www.mathworks.com, 2017. Accessed August 23, 2017.

[21] Website. Current readings for uw weather station. http://weather.uwaterloo.ca/, 2017. Accessed August 23, 2017.

[22] Website. Eigen. http://eigen.tuxfamily.org/index.php?title=Main_Page, 2017. Accessed August 23, 2017.

[23] Website. Nvida tensorrt. https://developer.nvidia.com/tensorrt, 2017. Accessed August 23, 2017.

[24] Website. Nvidia jetson tx1 supercomputer-on-module drives next wave of autonomous machines. https://devblogs.nvidia.com/parallelforall/nvidia-jetson-tx1-supercomputer-on-module-drives-next-wave-of-autonomous-machines, 2017. Accessed July 26, 2017.

[25] Website. Tensorflow main page. https://www.tensorflow.org/, 2017. Accessed June 6, 2017.

APPENDICES

Appendix A

Implementation Superclass of S-CNN Optimization

```
#pragma once

class ConvLayerImp {
 public:
    virtual ~ConvLayerImp() {}

    virtual int rows() const = 0;
    virtual int cols() const = 0;
    virtual int channels() const = 0;
    virtual int kernels() const = 0;

    virtual void calcConvolutions(const float* p, float* r) = 0;
};
```

Appendix B

Implementation Header of S-CNN Optimization

```
#pragma once
#include "conv_layer_imp.h"

class Conv1_1: public ConvLayerImp {
 public:
   Conv1_1() {}

   static const int kRows = 321;
   static const int kCols = 321;
   static const int kChannels = 3;
   static const int kKernels = 64;

   virtual int rows() const {return kRows;}
   virtual int cols() const {return kCols;}
   virtual int channels() const {return kChannels;}
   virtual int kernels() const {return kKernels;}

   virtual void calcConvolutions(const float* p, float* r);
};
```

Appendix C

Implementation Code of S-CNN Optimization

```
#ifndef GENERATE_CODE
#include <math.h>
#include "conv1_1_imp.h"

void Conv1_1::calcConvolutions(const float* p, float* r) {
   float pv[27];
   int i = 0;
   for(int y=0; y<kRows-2; ++y) {
      int ysj=kChannels*kCols*y;
      for(int x=0; x<kCols-2; ++x) {
         int xsj=kChannels*x;
         int j=xsj+ysj;

         pv[0]=p[j+0];
         pv[1]=p[j+3];
         pv[2]=p[j+6];
         pv[3]=p[j+963];
         pv[4]=p[j+966];
         pv[5]=p[j+969];
         pv[6]=p[j+1926];
         pv[7]=p[j+1929];
         pv[8]=p[j+1932];
```

```
            pv[9]=p[j+1];
            pv[10]=p[j+4];
            pv[11]=p[j+7];
            pv[12]=p[j+964];
            pv[13]=p[j+967];
            pv[14]=p[j+970];
            pv[15]=p[j+1927];
            pv[16]=p[j+1930];
            pv[17]=p[j+1933];
            pv[18]=p[j+2];
            pv[19]=p[j+5];
            pv[20]=p[j+8];
            pv[21]=p[j+965];
            pv[22]=p[j+968];
            pv[23]=p[j+971];
            pv[24]=p[j+1928];
            pv[25]=p[j+1931];
            pv[26]=p[j+1934];

            r[i]=0.856957f+0.473209f*pv[0]+0.562472f*pv[9]+0.447497f*pv[10]+
             ↪  -0.467997f*pv[14]+-0.521685f*pv[17]+-0.441567f*pv[23];
             ↪  r[i]=(r[i]>0?r[i]:0.0f); ++i;
            ++i;  // r[++i]=0.0379315;  // constant
            ++i;  // r[++i]=0;  // constant
            r[i]=1.07578f+-0.459466f*pv[7]+0.358358f*pv[10]+-0.538050f*pv[16
             ↪  ]+-0.432798f*pv[22]+-0.503271f*pv[25];
             ↪  r[i]=(r[i]>0?r[i]:0.0f); ++i;
            r[i]=0.200453f+0.307489f*pv[11]; r[i]=(r[i]>0?r[i]:0.0f); ++i;
            ++i;  // r[++i]=0.699564;  // constant
            ++i;  // r[++i]=0;  // constant
            ++i;  // r[++i]=0.448339;  // constant
            r[i]=0.406981f+-0.351085f*pv[17]; r[i]=(r[i]>0?r[i]:0.0f); ++i;
            ++i;  // r[++i]=0;  // constant
            /*More expressions*/
        }
    }
}
#endif
```