TAMPERE UNIVERSITY OF TECHNOLOGY

Renjie Xie

Dataflow-Based Implementation of Deep Learning Application

Master of Science Thesis

Examiner: Prof. Shuvra Bhattacharyya,
Prof. Jarmo Takala, and
Dr. Heikki Huttunen
Examiner and topic approved by the
Faculty Council of the Faculty of
Computing and Electrical Engineering
on Nov 4, 2015

ABSTRACT

TAMPERE UNIVERSITY OF TECHNOLOGY
Master's Degree Programme in Information Technology
Renjie, Xie: Dataflow-Based Implementation of Deep Learning Application
Master of Science Thesis, 51 pages, 3 Appendix pages
June 2016
Major: Pervasive Systems
Examiner: Prof. Shuvra Bhattacharyya, Prof. Jarmo Takala, Dr. Heikki Huttunen
Keywords: Dataflow, LIDE, DICE, Matlab, C, Deep Learning, DNN, Car Recognition

The proliferation of research on high efficient performance on deep learning has contributed to an increasing challenge and interest in the topic concerning the integration of this advanced-technology into daily life. Although a large amount of work on the domain of machine learning has been dedicated to the accuracy, efficiency, net topology and algorithm in the training and recognition procedures, the investigation of deep learning implementations in highly resource-constrainted contexts has been relatively unexplored due to the large computational requirements involved during the process of training large-scale network. In light of this, one process concentrated on parameters extraction and dataflow design, implementation, optimization of one deep learning application for vehicle classification on multicore platforms with limited numbers of available processor cores is demonstrated. By means of thousands of actors computation and fifos communication, we establish one enormous and complex dataflow graph, and then using the resulting dataflow representations, we apply a wide range of design optimizations to probe efficient implementations on three different multicore platforms. Through the incorporation of dataflow techniques, it is gratifying for us to see its effectiveness and efficiency in the several flexible experiments with alternative platforms that tailored to the resource constraints.

Besides, we pioneer three general, novel, primitive and thorough flow charts during the work - deep leanring model, LIDE-C establishing model, LIDE-C coding model. Finally, not only LIDE-C we utilize for the implementation, but also DICE we apply for validation and verification. Both tools are incubated by DSPCAD at Maryland of University, and will be updated better in the future.

PREFACE

This Master thesis is a joint project on efficient dataflow between DSPCAD GROUP at Maryland of University and the Department of Pervasive Computing at Tampere University of technology, part of which is supported by Tekes (Finnish Funding Agency for Innovation). The major aim is to contemplate a methodology and then implement an overall deep learning dataflow model through LIDE-C from scratch.

First of all, I would like to give my sincere thanks to my advisor and mentor Prof. Shuvra Bhattacharyya for his priceless guidance, support, encouragement and inspiration. His persisting support, not only made me own high confidence but also motivated me during the difficult time in my thesis. His introduction to the group members (Yanzhou Liu, Shuoxin Lin and Timo Viitanen) is to expand my friend circle as well as to further discuss on the topic, especially the tool DICE and LIDE-C tools developed by the DSPCAD GROUP. Due to his concentration on detail, thorough and disciplined review process, I learned much advanced and practical knowledge which, without doubt, will be fruitful and pay dividends in the future.

Furthermore, I also gratefully thank Prof. Jarmo Takala for introducing Prof. Shuvra Bhattacharyya to me, presenting me a brief of relative dataflow knowledge and also giving me an opportunity to conduct this topic, particularly at the beginning of my master thesis.

Finally, I would also express my appreciation to Dr. Heikki Huttunen for deep learning knowledge, structure, multicore platforms (merope instructions) and especially the introduction to his recent paper from the department of signal processing, which illustrates a clue where the topic starts and a vision on the meaning of my master thesis topic.

For all these and many all, I am so grateful and thankful to all of you. Certainly, three years studying in Tampere University of Technology broadens my knowledge span intensively and extensively, many thanks!

Tampere. 22.03.2016

Renjie Xie

CONTENTS

TERMS AND DEFINITIONS

BDF	Boolean Dataflow
BP	Back Propagation
CFDF	Core functional dataflow
CNN	Convolutional Neural Network
CSDF	Cyclo-Static Dataflow
CTC	Computation to Communication
DBN	Deep Belief Network
DICE	DSPCAD Interative Command Line Envirnment
DL	Deep Learning
DNN	Deep Neural Network
EIDF	Enable-Invoke Dataflow
FP	Forward Propagation
ILSVRC	Large Scale Visual Recognition Challenge
LIDE	Lightweight Datalow Environment
MLP	Multi-layer Perception
PD	Parameterized Dataflow
PSDF	Parameterized Sychronous Dataflow
RNN	Recurrent Neural Network
SDF	Synchronous Dataflow
SIMD	Single Instruction Multiple Data

1. INTRODUCTION

With the increment of appealing on smart city establishment all over the world, the increasing development of artificial intelligent learning in an academic circles, and the growth of associated applications in mobile and distributed contexts, big data and deep learning have been heating "words" (Fig. 1.1), however, it correspondingly brings a throng of questions – how to select the best deep learning model tailored to one specific application; how to training the net model with a view to increasing the prediction of overall performance and accuracy; and how to excavate and extract invaluable features from the nearly an infinite loads of datum, and so on.

It is admitted that a great deal of advanced-algorithm on machine learning has already been contributed and testified its feasibility, flexibility and adaptivity for a diversified of applications, but it is still impractical to apply these methods into our daily life without advanced-supercomputers. The reason for that is these algorithm set up a byzantine framework that is rooted in complicated computation, convoluted network structure and many layers' iterations. Correspondingly the space and time on operation are extremely demanding, which seems paradox to a broad spectrum of real-time application areas, like surveillance, intelligent transportation, particularly the application in the small smart devices, like IC chips, Cell-Phone, IPAD. In addition, with the concurrent advances in application areas for ubiquitous embeded computing, such as automotive embeded systems and the Internet of things, it also motivates the investigation of design methodologies for deploying deep neural network systems on resource-constrained embeded platform. All in all, future trend needs further simplification with regards for a trade-offs among DNN complexity, classification accuracy, real-time implementation performance, and resource requirement(cost).

1.1 Thesis Objective

The main objective of this thesis is to make an implementation on one deep learning application, employing vehicle classification as a case study to concretely demonstrate the methodology throughout the thesis and to sum up a series of methods that are developed to accommodate algorithm-, application-, implementation and design space-models and integrate them into a systematic manner for optimized system design.

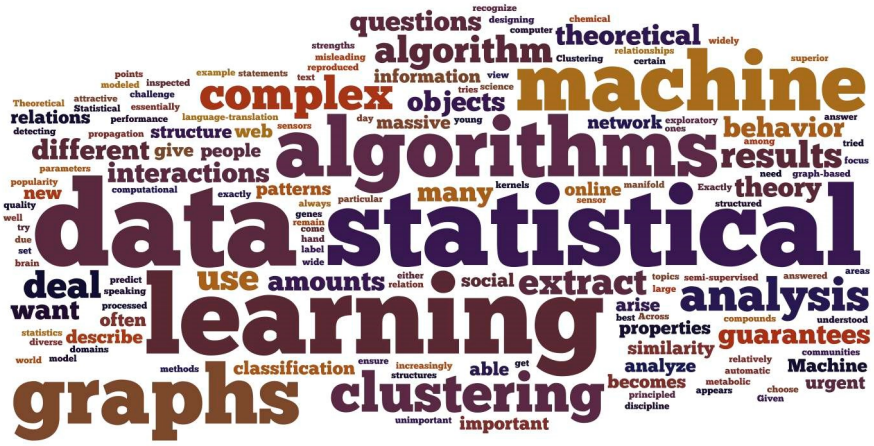


Figure 1.1: Heating words.

1.2 Author Contribution

We apply the signal processing oriented dataflow model of computation and communication and employ the resulting dataflow representations to implement, experiment with, iteratively optimize deep learning vehicle classification on three different multicore platforms using limited numbers of processing cores. More specifically, it introduces a unified methodology for modeling, mapping, and transforming deep learning implementations using dataflow technique, along with methods to integrate the hyperparameter tuning and simulation processes of deep learning system design with the proposed dataflow-based implementation approach. While this methodology is not specific to any particular application area, it is particularly well suited to embedded signal, image and video processing applications, where dataflow-based design is especially relevant. As a mentioned above, the thesis contribution is as following:

- Project Methodology – how to set up DNN to specific application and then breed up one product from zero (Fig. 8.1).
- LIDE-C Methodology – how to design, implement and optimize one DNN application in LIDE-C from scratch (Fig. 8.2).
- Code Methodology – how to write codes (actor, fifo and graph) based on dataflow model from nothing (Fig. 5.12).
- Parallel and distributed computing on LIDE-C model.

1.3 Thesis Organization

Chapter two and three provides background on two topics that are relevant for this research, separately the theory of deep learning and dataflow modeling. The rest of the thesis, one dataflow-based implementation of vehicle recognition application based on the recent of deep neural network [1] is demonstrated. Chapter four is to introduce how to select the hyper-parameters for the best deep learning network topology specific to one application, and then matlab simulation. Chapter five is the design and implementation on this application using lightweight dataflow technique, followed by the transformation on the dataflow graph described by chapter six, and The experimental results are recorded in the chapter seven. Finally, the conclusions and future work is presented in chapter eight.

2. DEEP LEARNING

In artificial intelligence, deep learning has attracted great research interests in many signal processing application areas recently and also gets numerous fruits from time to time. To start with, in this chapter, the related task of image classification is introduced. Next, some popular datasets used in recent years are listed and illustrated. At last, the basic theories and toolbox pertaining to the thesis are presented.

2.1 Image Classification

Image classification is one of the most fundamental application in deep learning domain, such as face recognition[2]. Through analysis and numerical property extraction from various image features, one label is attached to one input image within a predefined set of categories. The algorithm that maps a wide range of images to its corresponding categories is called classification, typically consisting of two phases of processing: training and testing phase shown on Fig. 2.1.

The target of initial training is to capture and isolate some salient properties of typical image features. On the basis of these, a special description of each classification category is created. The subsequent testing phase is to classify image feature from these feature-space partitions. There are two major categories techniques for image classification - supervised classification, where the training data are accompanied by the labels indicating the categories of the observations and then new data is classified based on the training set, and unsupervised classification, where the training data without the label naturally and automatically set up one groups of the similar features and new data is used in feature extraction, clusters and other purposes. The merit and demerit of both classification is illustrated in Fig 2.2, and the most significant benefit of unsupervised learning is there is no need for annotation.

2.2 Datasets

With the development of computer vision, a growing number of datasets are collected and emerged in order to meet various requirements and solve different kinds of image classification issues. Some popular dataset is described as following.

MNIST dataset [3] is a huge database of handwritten digits that has 60,000 training example and 10,000 testing example, commonly focused on the deformed

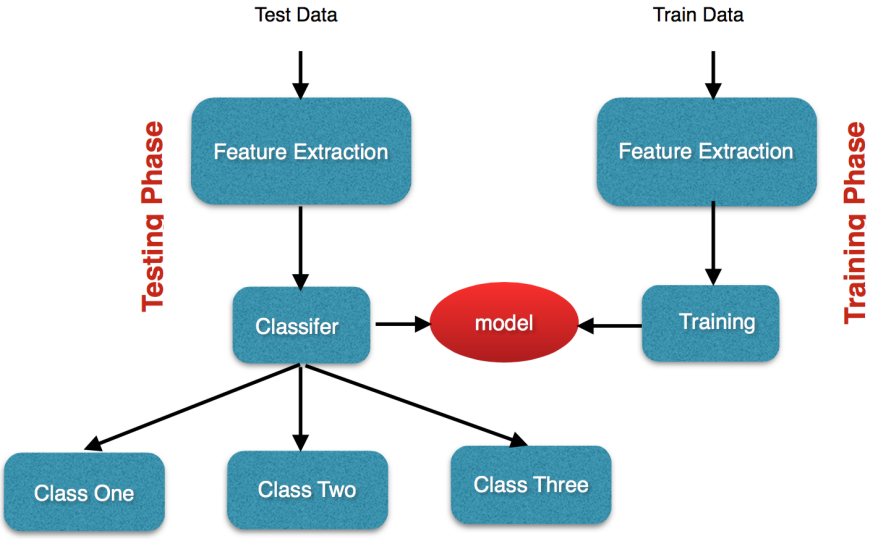


Figure 2.1: Training and testing phases.

image. Figure 2.3 shows the example of MNIST dataset.

CIFAR-10 dataset [5] possesses 60,000 natural images within 10 categories, averagely 6000 pieces of 32 x 32 RGB images per one categories. All datasets are decomposed into training data, Randomly selecting 5000 images from each categories, and testing data, the rest 10000 images of the datasets. Ten categories separately are airplane, automobile, bird, cat, deer, dog, frog, horse, ship and truck. Figure 2.4 shows the example from CIFAR-10 dataset.

CIFAR-100 dataset is the extension of CIFAR-10 dataset. It has 100 classes (fine label), each of which is composed by 500 training images and 100 testing images. Furthermore, 100 classes are fallen into 20 superclasses(coarse label). Each image is 32 x 32 RGB image with one fine label and coarse label.

ImageNet dataset[7] is a large-scale database organized based on the WordNet hierarchy. There are millions of images from more than 20, 000 categories, a very useful resource for image classification, object detection and localization. A large number of researchers in the academic field, as well as educators all over the world apply this dataset to participate two main competitions and two taster competitions

	Pros	Cons
Unsupervised Classification	- No prior knowledge of the region is required - Allows for minimisation of human error - Spectrally distinct areas presented which may not have been obvious to the human eye	- Spectral grouping may not correspond to information classes of interest to the analyst - Analyst has little control over the classes
Supervised Classification	- Analyst has control - Operator can often detect and rectify	- Collecting training data is time consuming - There is no way to recognise and represent categories which are not represented in the training data

Figure 2.2: Pros and cons of supervised and unsupervised classification.

Figure 2.3: Examples of MNIST dataset [4].

held by ILSVRC each year.

In this thesis, we need to draw a line between objects that belongs to one categories(vehicle). New database collected by company Visy Oy, which has gathered tens of millions of vehicle image over the year, finally tailored to 6555 pieces of 96 x 96 color images in four classes (car, van, bus, truck that belongs to vehicles). Paper [1] illustrates the details of the image collection.

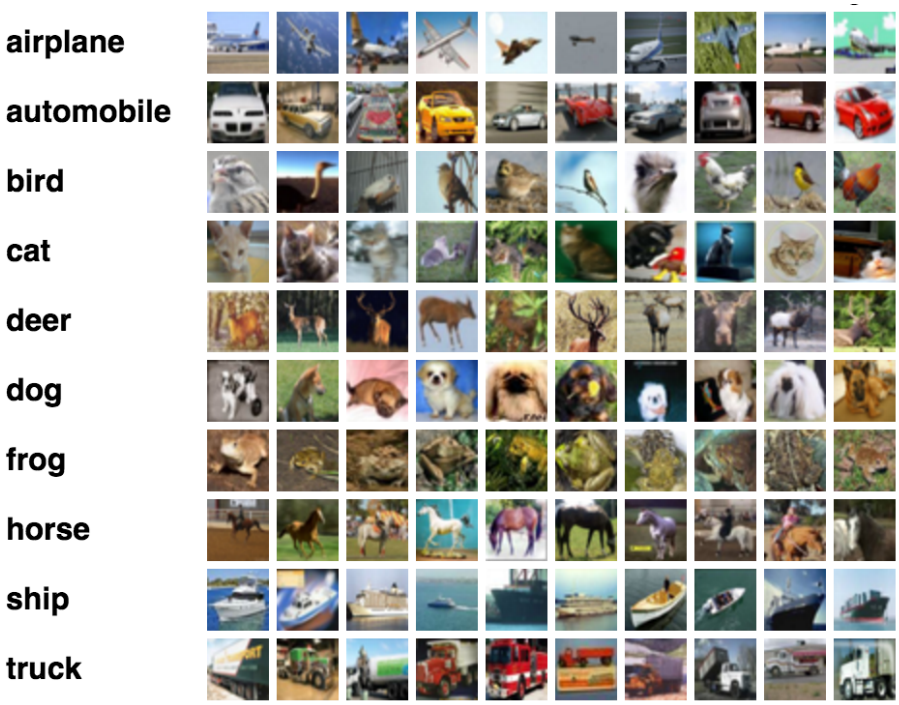


Figure 2.4: Examples from CIFAR-10 dataset [5].

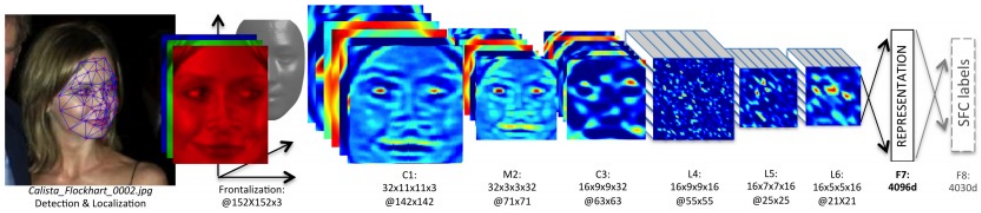


Figure 2.5: Features learned from training on deep face [6].

2.3 Neural Network and Deep Neural Network

The general idea of neural network is Sprung from biological neural network, analogically evolved into ICT field, a massively and complexly parallel distributed neural model that contains a great deal of neurons (processing unit), which are the fundamental to operation of a neural network, namely dealing with experimental knowledge(computation) and self-studying. Figure 2.5 demonstrates features learned from training on face recognition, which is robust to errors in the training process. Traditional neural network is multilayer perception(MLP)[8], back propagation(BP)[9], [10] and recurrent neural network(RNN) [11], in possession of various properties - nonlinearity, input-ouput mapping, adaptively and so on. Figure 2.6 shows the traditional neural network architecture.

After several years of development, however, deep neural network[12][13][14][15] still emerged because of these bottlenecks in the traditional neural network - the more layers structure in MLP does not work well as a result of diminishing error problem, which means the error propagating from the output layer to input layer is getting smaller and smaller (cannot learn), while only three layers MLP is only

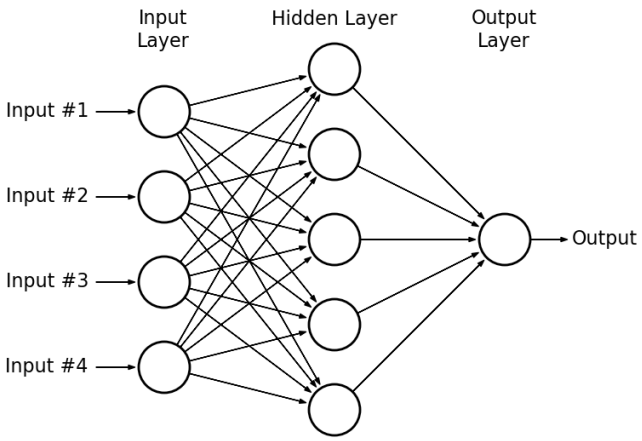


Figure 2.6: Traditional neural network.

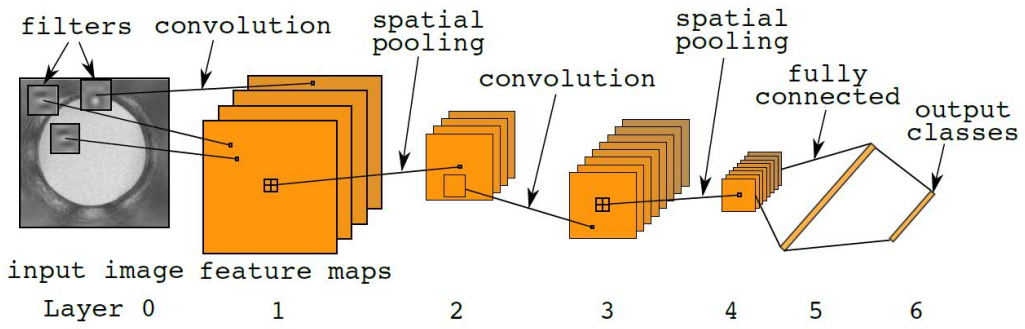


Figure 2.7: Deep neural network(CNN) [16].

a universal approximator. Deep learning solve this by layer-wise mechanism (learn from the lower layer before move up to the higher layers) and fine-tuning (adjust the unsupervisedly learn weights). Figure 2.7 shows deep neural network architecture, it is obvious that deep learning provide more abstraction complexity and hierarchical features learning through its more layers. Such is to lead to the currently best recognition performance of deep learning in many cases [17],[18],[19].

2.4 Convolutional Neural Network

Convolution neural network [20], [21] - a multiple-layers neural network, which is composed by several two dimensional surfaces containing several independent neurons, in possession of local receptive fields, shared weights, and the time or spatial sub-sampling, displacement , scale invariant deformation advantage(to some degree) - is one of the most popular and distinction in the deep neural network, especially in the computer vision filed. As the typical deep learning, CNN employs the feed-forward propagation for recoginition and backforward propagation for training. In the thesis, we train CNN off-line in the supercomputers and then used after-trained network to perform time-sensitive recognition. Therefore, the time consumption of

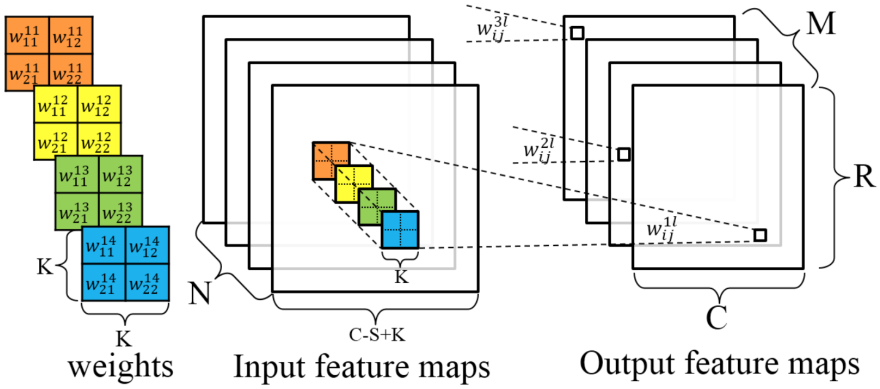


Figure 2.8: Graph of a convolutional layer [21].

feedforward propagation is what we focus.

There are two components in the CNN: feature extractors and a classifier. For one thing, the target of feature extractors is to filter the vectors of image datum into many same or lower dimensional vectors of "feature maps", separately representing various kinds of features - corners, lines, edge and so on. For another thing, the classifer is used to predict the maximum likelihood of predefined categories that input image belongs to. Figure 2.7 illustrates one example of convolutional neural network, which is composed by several feature extractors (two convolutional layers, two pooling layers, one dense layer) and final classifier layer. Among these, convolutional layers are accounted for more due to its high computation and most complexity. Figure 2.8 shows the example of one convolutional layer. There are N input feature maps, and each one makes convolutions with a shifting window (the size of K x K) to breed up one corresponding pixel in one specific output feature map (R x C). After the complete of the convolutional layer, the total of M output feature maps will be the set of next layer's inputs to do next operations. The recent study [21] on feedforward propagation proves that the computation time of the convolution operations will account for the 90% of the whole processing time. Therefore, the optimization described later will be concentrated into this point.

2.5 Deep Learning Toolbox - Caffe

Caffe[22], [23] is a deep learning tools developed by the Berkeley Vision and Learning Center (BVLC) and by community contributors. It is majorly composed by C++ libraries, python and matlab interfaces, provides a series of utilities for training, testing, fine-tuning and anything the designer could make use of during the process of research. The merit of the caffe is modularity - facilitate the design, modification and extension of datum, layers, functions and structures; expression - implement the designed network in one configuration file; reference model - apply pre-train model for research. Caffe has it own definition on the basis of layer-by-layer and bottom-to-top module from input data to loss. Blobs, layers and networks constitutes the caffe network. Figure 2.9 is the layer computation and communication in the left and an concrete example of caffe network in the right.

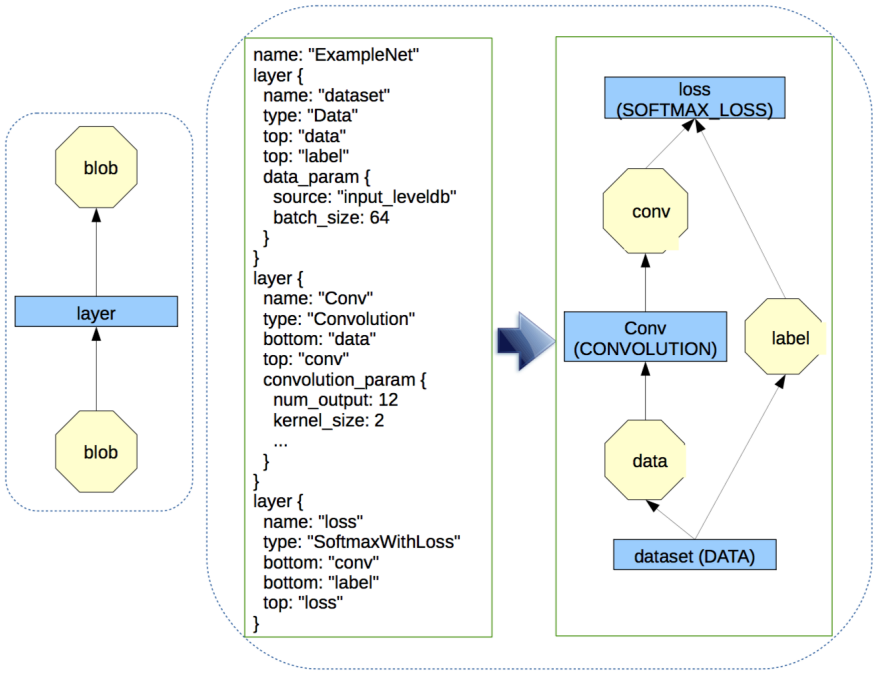


Figure 2.9: Left: layer computation and connections; Right: an example of Caffe network.

3. DATAFLOW MODELING

Dataflow-based model has been explored intensively and extensively over the recent years, especially in the embedded system due to the fact of difficulty in extracting the high level application structure from platform-based design tool, but dataflow model could facilitate system analysis, synthesis integration and optimization.

3.1 Dataflow Modeling Principle

In the context of dataflow modeling [24], a dataflow graph is represented as a directed graph, and composed by a set of actors(vertices) and a set of edges(first-in-first-out, FIFO), where the actors represent computational functions of arbitrary complexity and the edges represent communication channels between actors. Actors produce and consume data value, which is encapsulated in a token as it passes from the output of one actor to the input of another.

A dataflow edge could be represented as an order pair $e = (v_1, v_2)$ to mean data from v_1 to v_2. Here, v_1, denoted by $src(e)$, is called the *source actor* (or simply "source") of e, and v_2, denoted by $snk(e)$, is called the *sink actor* (or simply "sink") of e. One dataflow actor executes the graph by *enable* functions any time as long as sufficient data from its incoming edges is adequate to perform its specific computation, where each actor execution consumes and produces a well-defined number of *tokens* on each input and output port respectively. A dataflow graph $firing$ is a well-defined discrete units of execution.

In Fig. 3.1, FS1, FS2 are the actors of type “File Source”; Adder is an actor of addition operation; FK is an actor of type “File Sink”. The whole graph produces (consumes) one token onto (from) each actor output (input) port per actor firing.

3.2 Overview of Dataflow Models

There are several number of dataflow models that are applied into the design and implementation of DSP context.

Core Functional Dataflow (CFDF) [25] is a dataflow model of computation that is geared towards design, analysis, and implementation of signal processing system. CFDF can be regarded as a programming model for developing signal processing components and systems that have already known ratio of production and consumption, as well as ones that utilize dynamic dataflow rates.

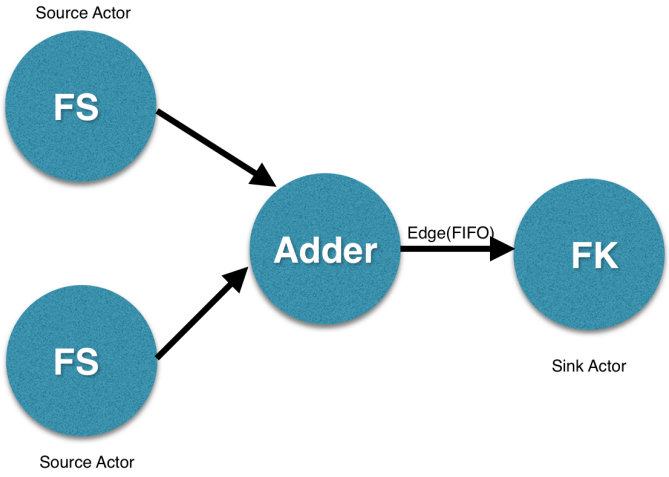


Figure 3.1: Simple dataflow graph.

Synchronous Dataflow (SDF) [26] introduced by Lee and Messerschmitt, is the simplest and most popular form of dataflow model, which imposes the restriction that the number if data values produced by an actor onto each outgoing edge is constant, and also similarly that the number of data values consumed by an actor from each incoming edge is constant.

Cyclo − Static Dataflow (CSDF) [27] is a generalization of SDF. In CSDF, the number of tokens produced and consumed by an actor is permitted to vary as long as the variation takes a fixed and periodic pattern.

Parameterized Dataflow (PDF) [28] is a meta-modeling approach for integrating dynamic parameters and run-time adaption of parameters in a structured way into a certain class of dataflow models of computations, in particular, those models that have a well-defined concept of a graph iteration.

Boolean Dataflow (BDF) [29] model of computation is the extension of synchronous dataflow with another class of dynamic actor, where the production-to-consumption ratio on one actor ports depends on two-valued functions of control tokens, which originates from one designated control ports in dynamic dataflow actors.

Enable − Invoke Dataflow (EIDF) [30] is another dynamic dataflow modeling technique. It divided actors into a set of modes, each of which has a fixed number of tokens consumed and produced, representing one branch/process that can be exploited during the switch of a diversified of modes at run time.

3.3 Dataflow Modeling Environment: LIDE-C

LIDE-C (lightweight dataflow environment C) is a flexible design and C programming environment that allows designers to excavate dataflow-based techniques for design, implementation and optimization of signal processing systems [31] [32].

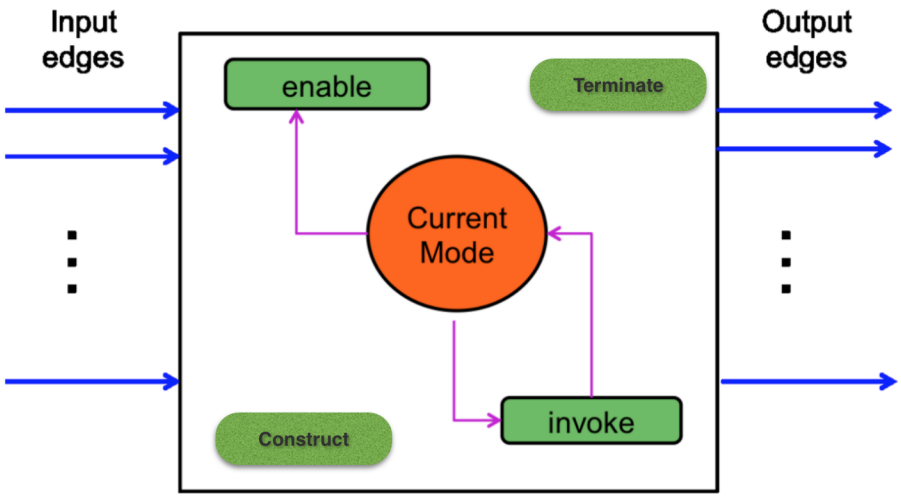


Figure 3.2: Actor interface function.

LIDE-C concentrates on essential application programming interface (API) features for signal processing oriented, dataflow-based development. The whole framework provides capabilities for implementing signal processing systems in a wide range of programming languages, and across a broad spectrum of platforms, including field programmable gate arrays (FPGA), graphics processing units (GPU), desktop workstations, and programmable digital signal processors.

LIDE-C software package possesses a number of libraries of dataflow graph element(actor and edge) implementations. Based on these basic elements, designers can freely design their own dataflow graph and define elements, develop specific-application (e.g, control-, parameterization-, and instrumentation-related modules), and schedulers that fires the whole dataflow graph sequentially. The details on installing the LIDE-C environment can be found from [33].

As described in chapter 3.1, two components - actors and fifos, are key points in dataflow model. For one thing, actor design in LIDE-C includes four interface function: the construct, enable, invoke and terminate functions (Fig. 3.2). The creation and definition of one actor in LIDE-C is the realization of four interface implementation:

(1) Construct Function: to create an instance of the actor and connect the ports of actor to a set of edges that is passed through the function argument list.

(2) Enable Function: to check at run time whether or not a given actor is firable - whether there is enough input data and empty buffer space to support the next firing of the actor.

(3) Invoke Function: to perform a single firing/block of firing for the actor.

(4) Terminate Function: to close out aspects of the underlying actor, including deallocation of relevant storage objects, once the actor is no longer needed in the context of its enclosing graph.

For another thing, FIFO design for dataflow graph implementation in LIDE-C is orthogonal to the design of dataflow actors. That means application designers can concentrates on design of actors(like algorithm) and then integrating these actors through well-defined interfaces and fifos. The beauty of this is to separately center around computation and communication with actor and fifo implementation. FIFO operations are encapsulated by interface functions in C. Function pointers are applied to point towards these interface functions so that they could be targeted to different implementation in the different forms while attached to the standard interface. Standard FIFO operations in LIDE-C execute the following tasks [34]:

(1) Create a new FIFO with a particular capacity.

(2) Read and Write tokens from/to one fifo.

(3) Check the capacity of the FIFO.

(4) Check the number of tokens that are currently in the FIFO.

(5) Deallocate the storage with the FIFO after the complete of using the FIFO.

After the creation of all actors and fifos in one dataflow-model graph application, gradually connecting and firing the graph one step by step is the next key point to verify the whole graph in the complex topology. Chapter five demonstrates one example based on car recognition application.

4. APPLICATION AND SIMULATION MODEL

The deep learning application that we focus on in this thesis is that of image-based recognition of vehicles. In particular, we develop DNN implementations for automatic discrimination among four types of vehicles — bus, truck, van and car. First step is to build on the DNN network structure based on DNN-based vehicle classification. Next, we implement the DNN system in MATLAB for simulation and testing purposes. The primary objective of this step is to collect results from each layer so that the embedded implementation for each layer can be tested in isolation in addition to performing complete, system level tests of the target implementation. Such layer-by-layer testing helps to build up the implementation incrementally, and localize the causes of test failures to provide for more rapid design iterations.

4.1 DNN Topology for Vehicle Classifier

Deep learning tool "Caffe" is applied into this application, to randomly search for the best combination of selected hyper-parameters (the number of layers, nodes, the size of convolutional kernels, etc). After a series of iterations on fifty pieces of random hyper-parameters with the same computational resources, the hyper-parameters for one of the best deep learning network topology is summed up to Table 4.1. The rightmost column (Selected Value) tabulates the relative parameter of deep learning network.

Description of CNN's structure:

(1) Two convolutional layers + two dense layers + one classifier layer

(2) 96 x 96 square pixels size of input image.

Hyper-parameter	**Range**	**Selected Value**
Number of Convolutional Layers	1-4	2
Number of Dense Layers	0-2	2
Input Image Size	64, 96, 128, 160	96
Kernel Size on All Convolutional Layers	5, 9, 13, 17	5
Number of Convolutional Maps	16, 32, 48	32
Learning Rate	10^{-5} - 10^{-1}	0.001643

Table 4.1: Hyper-parameters randomized over the iteration [1].

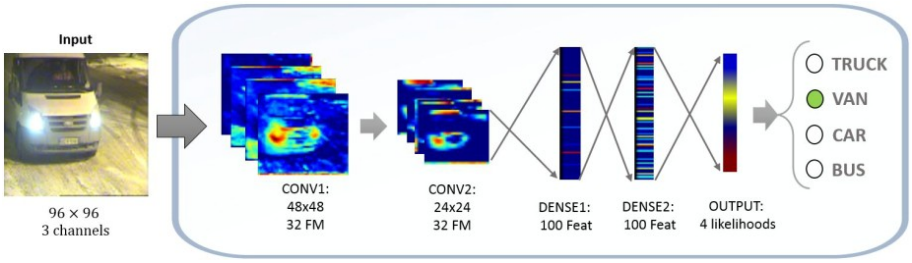


Figure 4.1: The structure of the proposed network [1].

(3) 5 x 5 square pixels size of convolutional kernels.

(4) 32 feature maps → 32 filters learned at each convolutional layer.

(5) 0.001643 learning rate for stochastic gradient back-propagation.

The diagrammatic sketch map on this CNN with these parameters is illustrated in Fig. 4.1. Broadly generalizing, the deep learning net topology[1] consists of five layers: two convolutional layers followed by two dense layers, plus an output layer. The first convolutional layer maps one input image of 96 x 96 R.G.B three channels into 32 pieces of feature maps, which are maxpooled and then Relu (Rectified Linear Unit) to 48 x 48 resolution. The second convolutional layer remaps another 32 pieces of 24 x 24 resolution's feature maps through the same functions, but the input is the 32 output feature maps from the first convolutional layer one. After two convolutional layers, the 32 output feature maps are fully connected into two dense layers with 100 nodes (features) each, and between layers there is an additional Relu non-linearity. Eventually, the outputs of two fully-connected dense layer runs to the last dense layer and then classified into one class with high probability among four classes by means of a softmax algorithm. he network is trained with a database of 6555 vehicle images and adjust the net parameters. After the experiment, the resulting prediction accuracy is 97.75%, which is clearly superior to the accuracy of earlier studies that use manually engineered feature extraction pipelines.

4.2 Parameters Extraction

This step is crucial to put the application on some less power but smart device, especially for so-far technology, because parameter extraction could save the training phase in the smart device and directly set up the specific system. Figure 4.2 reveals parameters conditions in each layer.

(1) The first and second convolutional layer respectively consist of 32 x 3 x 5 x 5 = 2400 and 32 x 32 x 5 x 5 = 25600 double type of values.

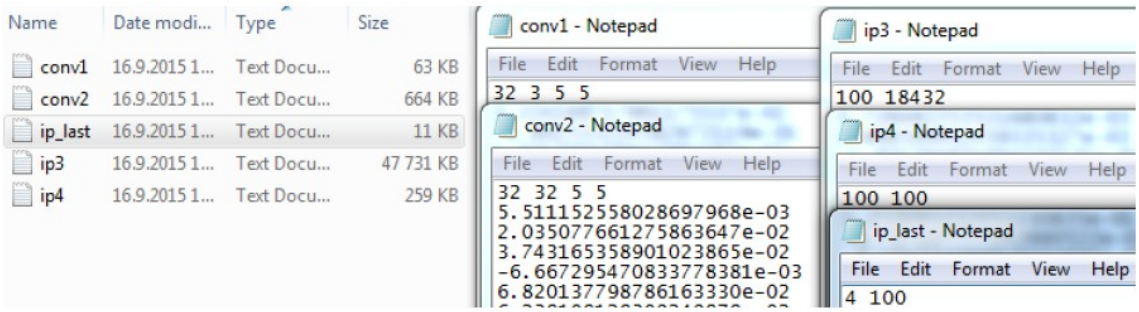


Figure 4.2: Parameters extraction.

(2) The number of the third layer's parameters is astonishing (exactly 100 x 18432 = 1843200 double type of values), and the size of the file is nearly up to 48M.

(3) The fourth and fifth dense layers has separately 100 x 100 = 10000 and 4 x 100 = 400 double type of values.

Although the amount of the datum is still gigantic, the consumption of time on loading is by far advantageous over the time on training phase.

4.3 Matlab Implementation

Figure 4.3 demonstrates the situation on matlab implementation. The key point of that is the utilization of cell array, which is involved with four-dimensional matrix and execute a series of operations within this huge four-dimensional space.

The description of figure 4.3 is as following:

- Directory "KERNELS-2015-05-23-12-53-32" contains all the figure **??** parameters' files.
- LoadLayer.m: load parameters of each layer to the variable as the form of cell array (4 dimensional matrix in the first two layers and 2 dimensional matrix in the rest of layers).
- Convolve.m: execute convolution operation in first two layers with the same the size of input and output. In terms of the number of input, there are three-channel inputs of the first layer while there are thirty two channels inputs of the second layer.
- Maxpool.m: occur in the first two layers, followed by the convolution operation. The maximum of value is picked up in every moving 2 x 2 square matrix (moving distance is 2).
- Relu.m: happen in the end of every layer to do with non-linear function.

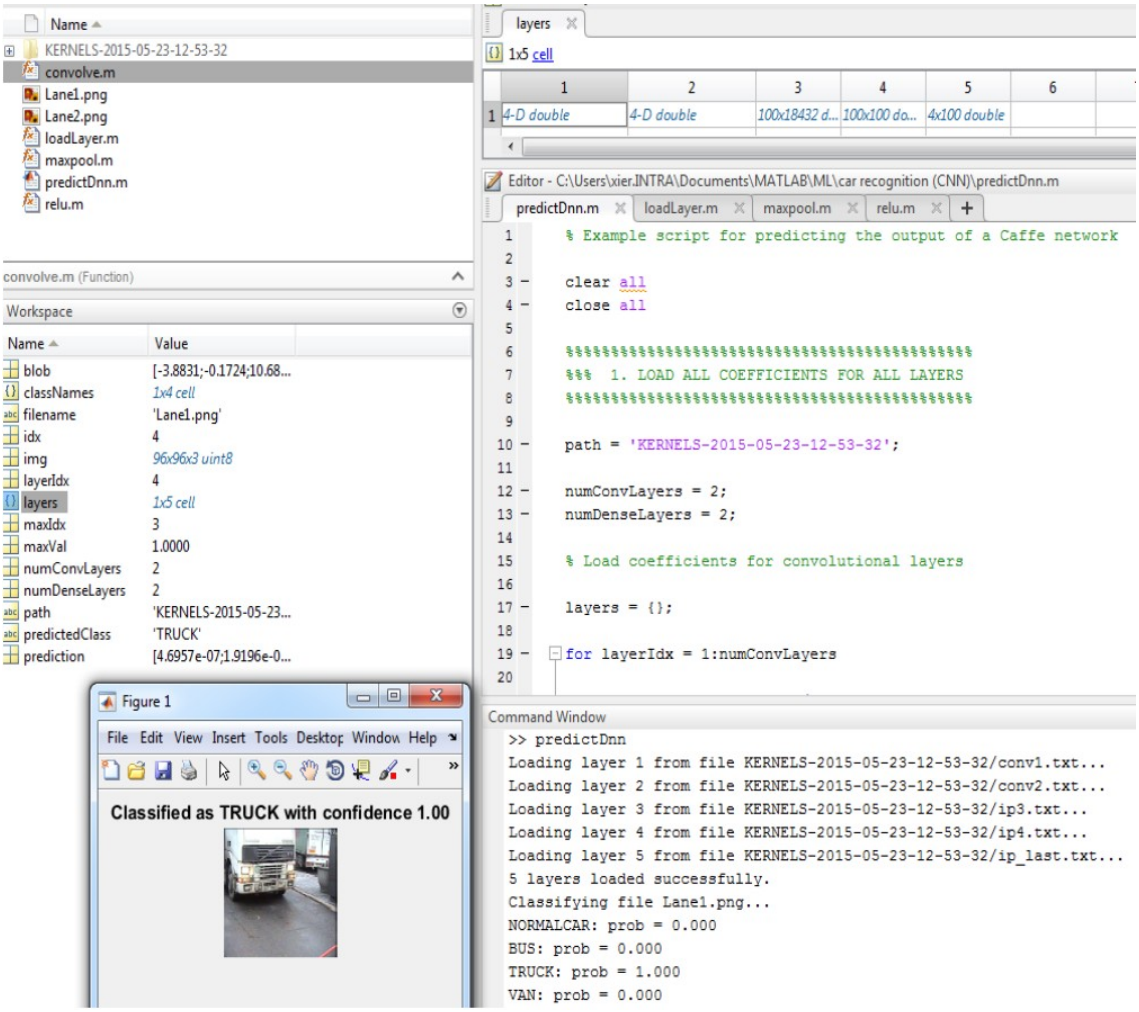


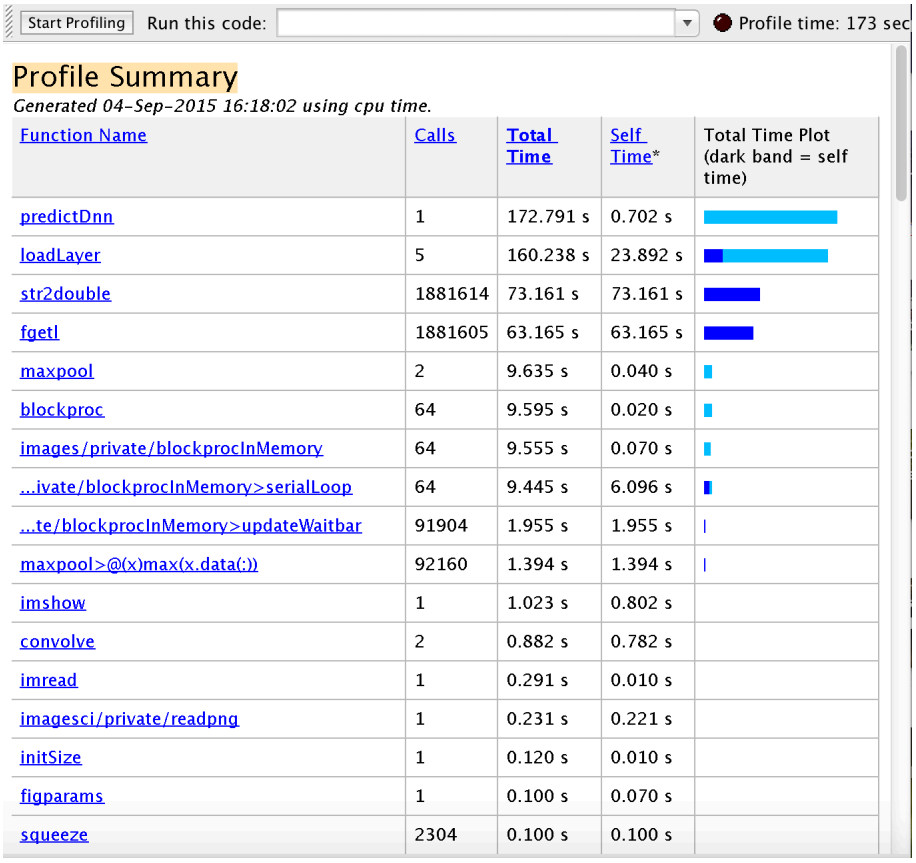
Figure 4.3: The implementation of matlab model.

- PredictDnn.m: the major car-recognition script that is to establish a CNN with the specific hyper-parameter, do operations, and then predict the result.

4.4 Experiment and Result

The criteria for testing the performance is the whole processing time , which is the sum of loading coefficient time and computing time. I put several experiment in different computers and get different results. Among this, the best result is loading coefficient is 50.51s and computing time is 1.36s, but the average consumption time is approximately 90s, The MATLAB Profiling tool's result is Figure 4.4.

From the Profile Summary view, the whole processing time is a bit longer (173s), which is beyond human-being's endurance, impractical to the real-time application. The reason for that is the utilization of four-dimensional cell array, which needs load, do and store four-dimensional cell array in every operation, wastes lots of processing time.



Profile Summary

Generated 04-Sep-2015 16:18:02 using cpu time.

Function Name	Calls	Total Time	Self Time*	Total Time Plot (dark band = self time)
predictDnn	1	172.791 s	0.702 s	
loadLayer	5	160.238 s	23.892 s	
str2double	1881614	73.161 s	73.161 s	
fgetl	1881605	63.165 s	63.165 s	
maxpool	2	9.635 s	0.040 s	
blockproc	64	9.595 s	0.020 s	
images/private/blockprocInMemory	64	9.555 s	0.070 s	
...ivate/blockprocInMemory>serialLoop	64	9.445 s	6.096 s	
...te/blockprocInMemory>updateWaitbar	91904	1.955 s	1.955 s	
maxpool>@(x)max(x.data(:))	92160	1.394 s	1.394 s	
imshow	1	1.023 s	0.802 s	
convolve	2	0.882 s	0.782 s	
imread	1	0.291 s	0.010 s	
imagesci/private/readpng	1	0.231 s	0.221 s	
initSize	1	0.120 s	0.010 s	
figparams	1	0.100 s	0.070 s	
squeeze	2304	0.100 s	0.100 s	

Figure 4.4: The profiling of matlab model.

5. LIDE C-BASED DESIGN AND IMPLEMENTATION

After developing the MATLAB-based simulation model for our DNN-based vehicle classification system, we proceed to develop an initial dataflow-based implementation, which will be employed as a starting point to evaluate the system on different kinds of platforms, and then iteratively optimize dataflow graph for the purpose of the improvement on performance.

5.1 Actors in Dataflow Graph

To begin with, the diagrammatic sketch map (Fig. 4.1) is transformed into the block diagram, which is depicted into the Fig. 5.1. Although this block diagram encompasses thousands of individual signal processing blocks (actors), there is a great deal of regularity in the way the blocks are instantiated and connected. Such regularity can be exploited from deriving LIDE-C designs in the form of compact, parameterized dataflow graph implementations that designers can efficiently analyze and manipulate (e.g., see [35]). The block diagram in Figure 5.1 incorporates a total of 10 different types of actors, which are summarized as following.

- Read Channel Actor: One image is decomposed into R.G.B three channels, and every channel has 96 x 96 resolution matrix to be read into one fifo.

- Covolutional Actor: A way of "multiplying together" two arrays of numbers to produce a third array of numbers with the same size and dimensionality. The formula definition is (5.1)

$$y[m,n] = \phi(p) = \phi(b + \sum_{k=0}^{K-1}\sum_{l=0}^{K-1} V[k,l]x[m+k,n+l]) \tag{5.1}$$

- Maxpool Actor: A form of non-linear down-sampling, which partitions the input image into a set of non-overlapping rectangles and, for each sub-region, outputs the maximum value. Figure 5.2 is the illustration of maxpooling definition. Left figure $\rightarrow$ the input volume of size [96 x 96 x 32] is pooled to the output volume of size [48x48x32] with the stride size 2, noted that the

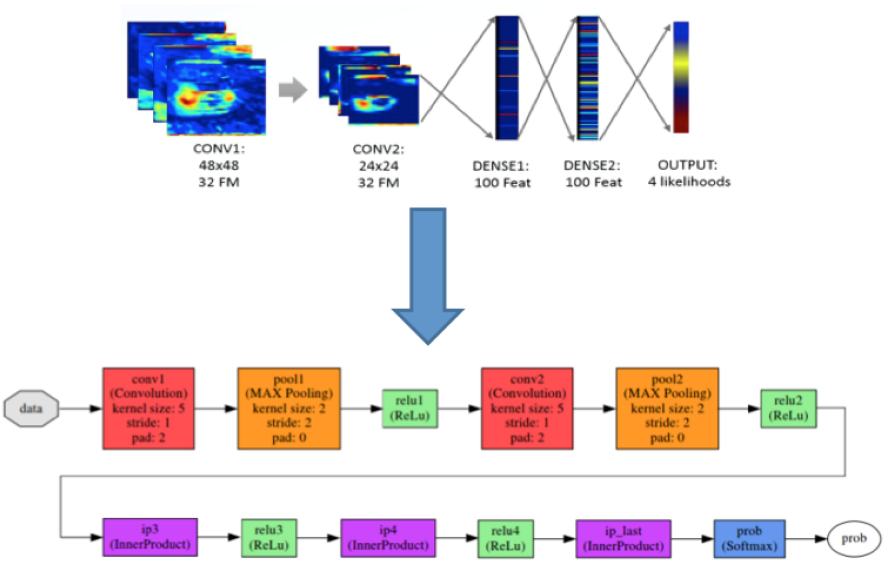


Figure 5.1: Block diagram of deep neural network.

depth is preserved. Right figure → the demonstration of maxpooling operation with stride 2.

- Relu Actor: Rectified Linear Unit, which is a very popular non-linearity function ⇒ $f(x) = max(0, x)$, x is the input. Softmax Actor: A neural transfer function, which calculates a layer's output from its net input ⇒ $a = exp(n)/sum(exp(n))$.
- Write Actor: Write the datum into the appointed file.
- All_to_one Actor: to decrease the dimension of the matrix, which means to assembly several inputs matrice to one output matrix. For example, 3 pieces of the size of 2 x 24 inputs would be assembly into one input matrix, whose size is 1728 x 1.
- Broadcast Actor: Copy the input matrix on every fifo outputs.

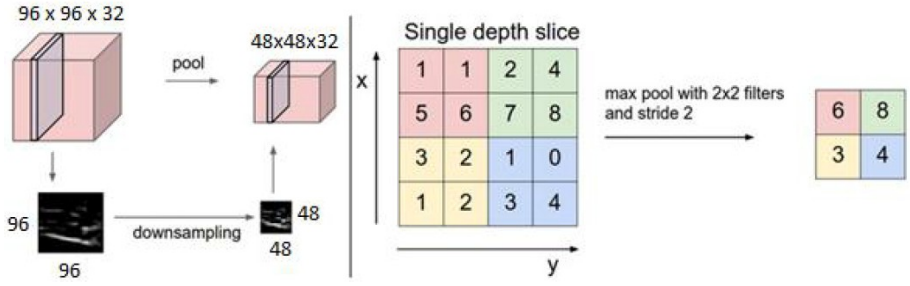


Figure 5.2: Maxpooling operation.

$$\mathbf{A} = \begin{pmatrix} A_{11} & A_{12} & \cdots & A_{1m} \\ A_{21} & A_{22} & \cdots & A_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ A_{n1} & A_{n2} & \cdots & A_{nm} \end{pmatrix}, \quad \mathbf{B} = \begin{pmatrix} B_{11} & B_{12} & \cdots & B_{1p} \\ B_{21} & B_{22} & \cdots & B_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ B_{m1} & B_{m2} & \cdots & B_{mp} \end{pmatrix}, \mathbf{AB} = \begin{pmatrix} (\mathbf{AB})_{11} & (\mathbf{AB})_{12} & \cdots & (\mathbf{AB})_{1p} \\ (\mathbf{AB})_{21} & (\mathbf{AB})_{22} & \cdots & (\mathbf{AB})_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ (\mathbf{AB})_{n1} & (\mathbf{AB})_{n2} & \cdots & (\mathbf{AB})_{np} \end{pmatrix}$$

$$(\mathbf{AB})_{ij} = \sum_{k=1}^{m} A_{ik} B_{kj}$$

Figure 5.3: The definition of matrix multiplication.

- Matrix Multiplication Actor: Assuming A is an n x m marix and B is an m x p matrix, the definition is Fig. 5.3, where each i,j entry is given by multiplying the entries A_{ik} (across row i of A) by the entries B_{kj} (down column j of B), for k = 1,2,. . . ,m, and summing the results over k.
- Matrix Addition Actor: Two matrices of equal number of rows and columns are added. The definition is Fig. 5.4, the sum of A and B is denoted A + B, which is computed by adding corresponding elements of A and B.
- Matrix Multiple Addition Actor: several inputs matrice with the same size and dimension add together once.

5.2 Design Dataflow Graphs

Though the whole CNN dataflow graph is not only much too huge but also little bit complicated and links is magnificently massive, the complex of network is majorly focus on the two convolutional layers that has some inherently regularity. Therefore, how to design the subgraph and then gradually establish the whole graph is a key point. In this section, we develop three different kinds of CNN design graph with different subgraph patterns. And every design has its own advantage and disadvantage.

5.2.1 Dataflow Model of Design One

The dataflow graph of design one is illustrated in figure 5.5.

- Five columns composed by different actors represents five layers of CNN.
- Every actor in Figure 5.5 is a hierarchical actor, which encapsulates one subgraph.

$$\mathbf{A}+\mathbf{B} = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix} + \begin{bmatrix} b_{11} & b_{12} & \cdots & b_{1n} \\ b_{21} & b_{22} & \cdots & b_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ b_{m1} & b_{m2} & \cdots & b_{mn} \end{bmatrix} = \begin{bmatrix} a_{11}+b_{11} & a_{12}+b_{12} & \cdots & a_{1n}+b_{1n} \\ a_{21}+b_{21} & a_{22}+b_{22} & \cdots & a_{2n}+b_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1}+b_{m1} & a_{m2}+b_{m2} & \cdots & a_{mn}+b_{mn} \end{bmatrix}$$

Figure 5.4: The definition of matrix addition.

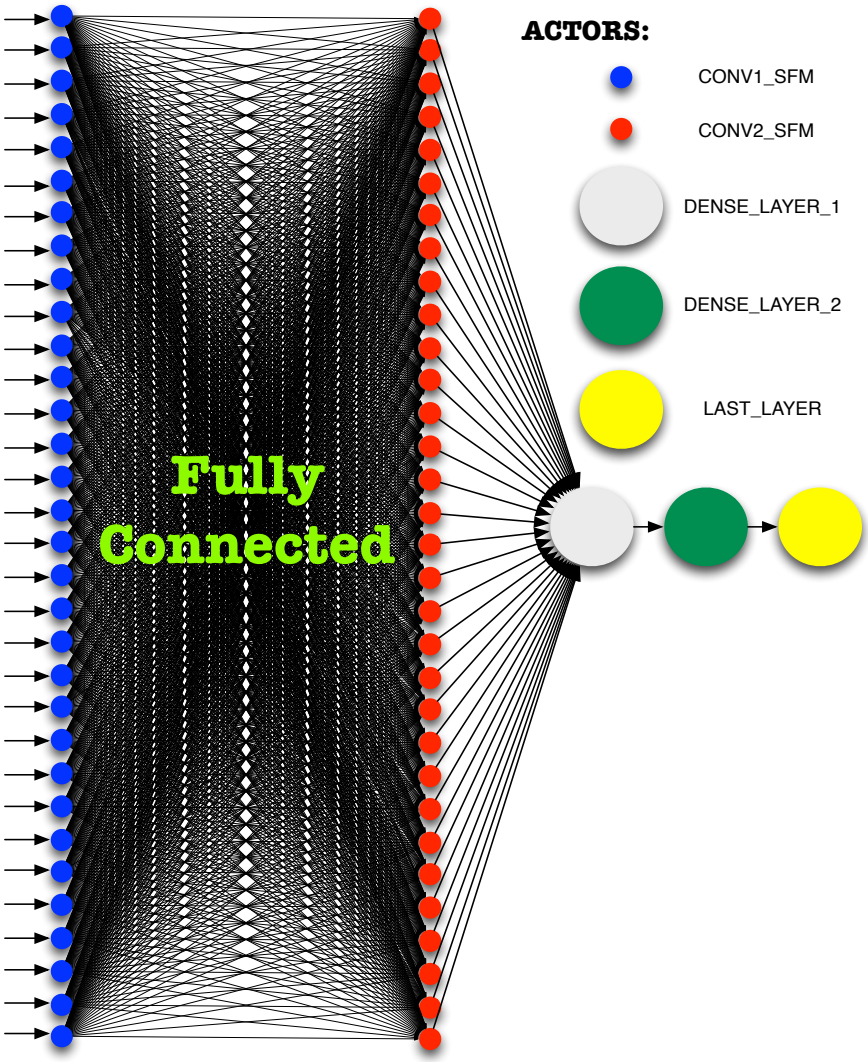


Figure 5.5: Dataflow model of design one.

- The first layer is made up of 32 conv1_SFM_Actors(blue), which signify 32 feature maps. The size of [3 x 96 x 96] matrix input maps to the size of [48 x 48] matrix output. Figure 5.6 is the details on subgraph_conv1_SFM.
- From convolutional layer one to layer two, every subgraph_conv2_SFM is fully connected all the outputs from subgraph_conv1_SFM, which means all the outputs of 32 feature maps from convolutional layer one is the inputs of every feature map in the convolutional layer two.
- The subgraph_conv_SFM actors(red) is almost the same function as subgraph_conv1_SFM actors(blue) - the symbols for the feature map in the second convolutional layer. Regard 32 outputs of the first convolutional layers as 32 inputs [48 x 48 x 32], which are transferred to the corresponding convo-

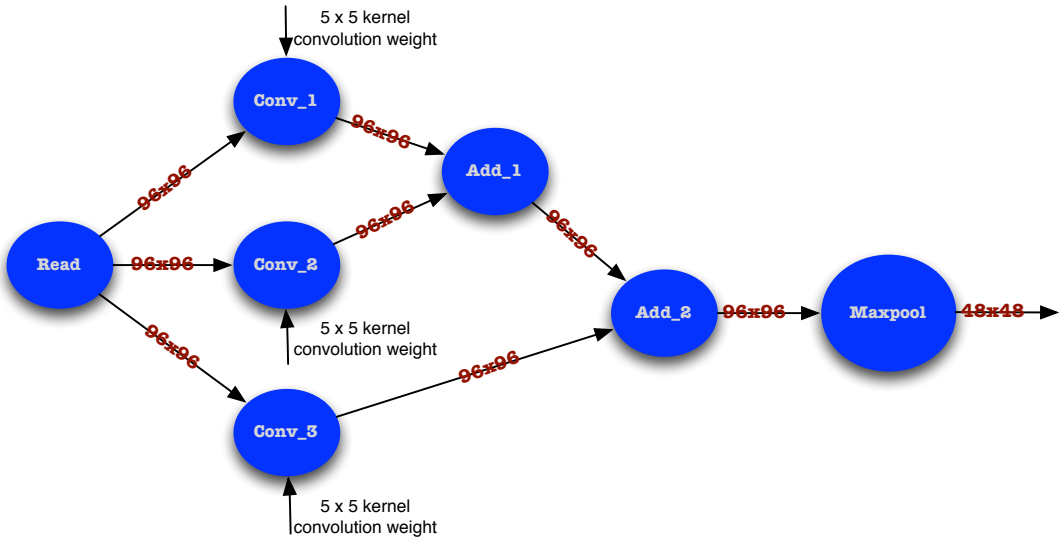


Figure 5.6: Dataflow subgraph of subgraph_conv1_SFM actor.

lutional actors, followed by adding all 32 output together and finally maxpool to [24 x 24] matrix output. The dataflow of subgraph_conv2_SFM is figure 5.7 .

- Dense_layer_one is composed by matrix multiplication actor and Relu actor. Assembly 32 pieces of [24 x 24] matrix into the size of [1 x 18432] before multiply the size of matrix [18432 x 100], and then go through Relu operation to get the final [100 x 1] size of output. Figure 5.8(a) is the dataflow of dense layer one .

- Dense_layer_two is also made up of matrix multiplication actor and relu actor. The difference from dense_layer_one is only one input fifo and the multiplication size of matrix is [100 x 100]. Figure 5.8(b) is the dataflow of dense layer two .

- The classifier_layer is a combination of matrix multiplication actor and softmax actor. By means of matrix multiplication [100 x 4], the matrix result is precisely classified into one of four results after softmax actor. Figure 5.8(c) is the dataflow of classifier layer.

Summary: The benefit of design one (based on feature map or layer as one actor) is that the whole network establishment structure is very crystal-clear, by far closest to the block diagram. Furthermore, it is straightforward to implement, validate and check the result as a whole graph. However, the drawback is the difficulty of further and deep optimizing when the subgraphs are determined, because one subgraph could be considered as a complete and "big" actor which is encapsulated, generally

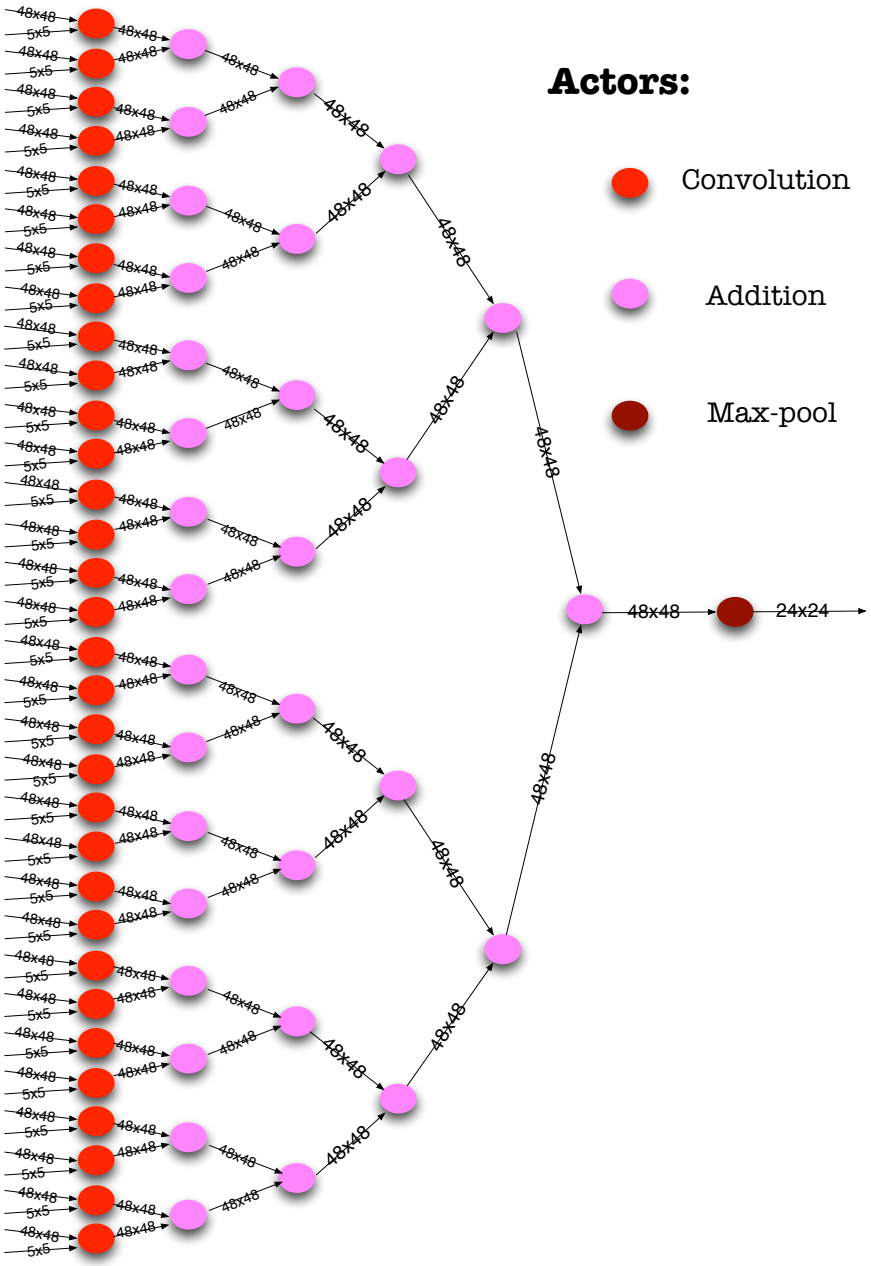
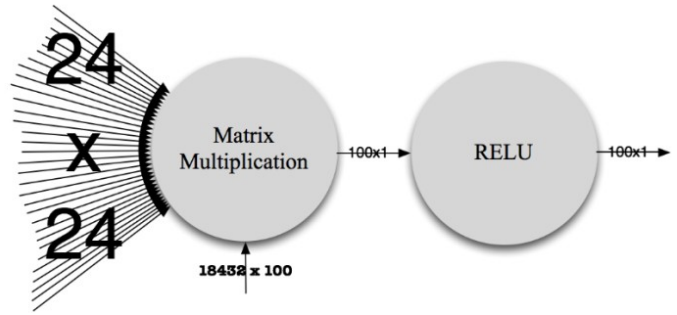
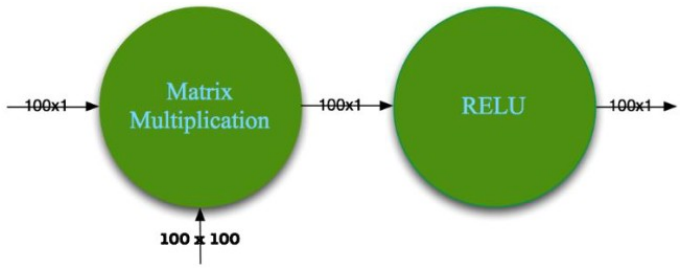


Figure 5.7: Dataflow subgraph of subgraph_conv2_SFM actor.

less to modify. Last but not the least, some subgraphs referred from other dataflow graph is not convenient to debug and check when it is encapsulated. Therefore it would be disaster if the error was occurred in the huge subgraph referred from other project when you had established tremendously large graph. All in all, it is a better method from scratch but more attention to the reference of other subgraphs written by other projects, more emphasis on the pre-requirement of actor.



(a) Dense layer one.



(b) Dense layer two.

(c) Classifier layer.

Figure 5.8: Layer three, four and five of the deep neural network.

5.2.2 Dataflow Model of Design Two

The concept on design two is the further decomposition of the first two convolutional layers into some basic and characteristic chunks. Figure 5.9 is the dataflow of design two.

The description of design two

- Sub_graph_1 (red) consists of two convolution actors and one addition actor, and the dataflow of sub_graph_1 is figure 5.10(a)
- Sub_graph_2 (blue) is short of one convolution actor compared to sub_graph_1, the dataflow of sub_graph_2 is figure 5.10(b)
- Maxpool actor for itself is one individual subgraph, the dataflow of maxpool is figure 5.10(c)

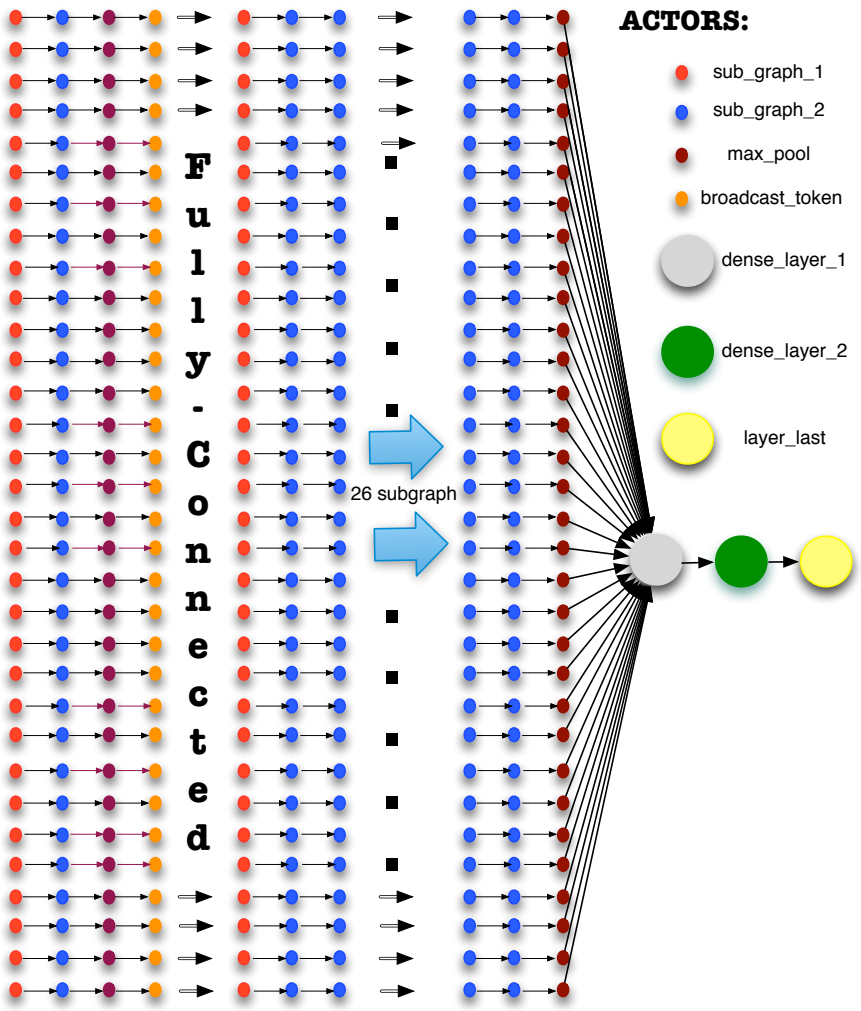
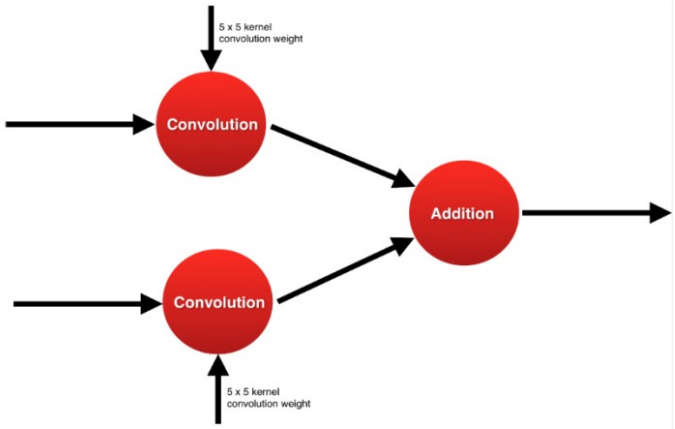


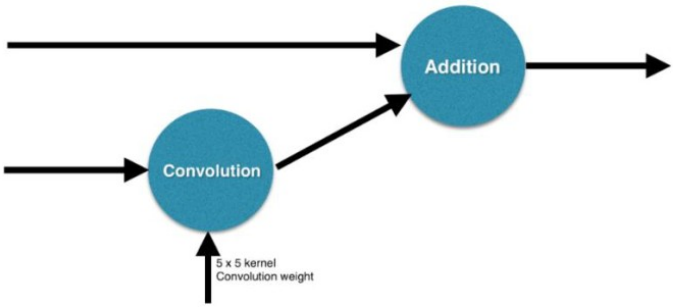
Figure 5.9: Dataflow graph of design two.

- Convolutional layer is made up of these three actors: run sub_graph_1 first, and iteratively operating several times of sub_graph_2 operation according to different layers, finally do maxpool.

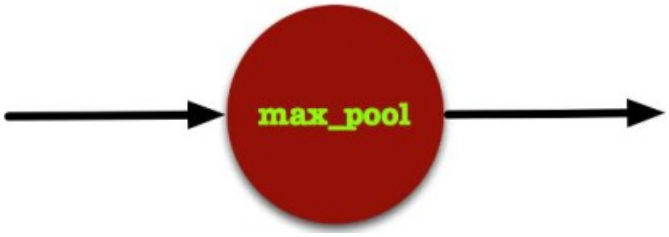
Summary: The advantage of the design two is that the dataflow graph is more clear to use loop unrolling (computing in parallel) and pipeline for optimization in the convolutional layer. That means the whole graph might extend the latency but increase the throughput. This feature is benefit to training the net (especially thousands of batch samples execute in the network and adjust the parameter through the forward, backward propagation). Certainly, it needs some tricks and analysis on the specific issue, and here it only comes up with possible idea. Overall speaking, the whole CNN dataflow graph on design two is still understandable.



(a) Sub_graph_1.



(b) Sub_graph_2.



(c) Maxpool_graph.

Figure 5.10: Sub_graph in Design Two.

5.2.3 Dataflow Model of Design Three

Although design two has already anatomize the whole graph to some degree, it does not dig deepest. The best and complete optimization, verification and debug always happen in the origin way. Therefore, the concept of design three is to consider one actor as one subgraph, which could divide the task into two aspects. On the one hand, actor is principally and exclusively the optimization of algorithm; on the other hand, how to optimize the whole graph is the scheduler responsibility. The dataflow of design three is the figure 5.11, but the huge graph is a little bit rearrangement and omitted in order to be readable and extendable in one page.

The description of design three:

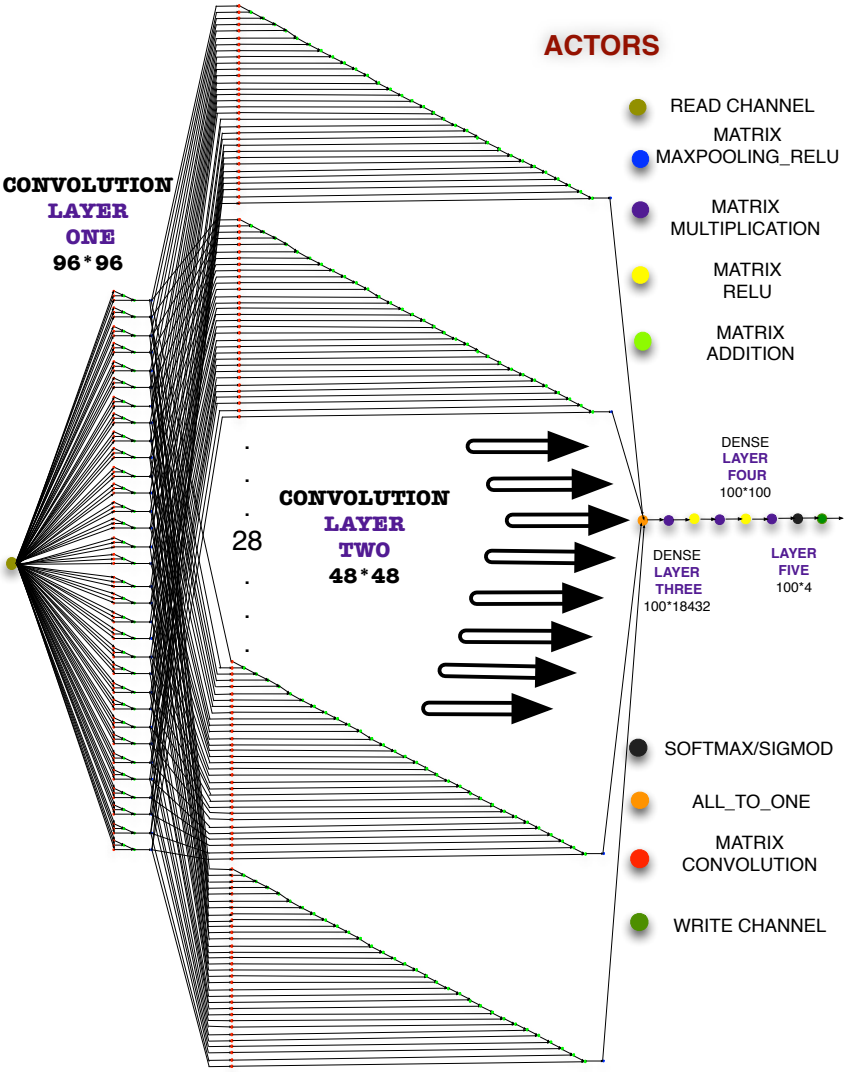


Figure 5.11: Dataflow graph of design three.

- Eight different actors are in the whole graph, and corresponding eight different subgraphs connects to the whole dataflow graph.
- Original convolutional neural network dataflow topology.

Summary: The merit of design three is to arbitrarily monitor, control, manage, validate and check the datum/buffer in every step/actor/area, and it is general to have inspiration of optimization from the primitive graph. Besides, do the parallel computing thoroughly (feature maps, loop unrolling in every convolutional layer; convolution actors parallelize in every feature map; convolution itself and matrix multiplication itself). Finally, it could simply execute some surgeons (optimization) in tiny region. Conversely, the demerit is that it is not simple to establish, optimize and implement the huge and complicated dataflow graph from start.

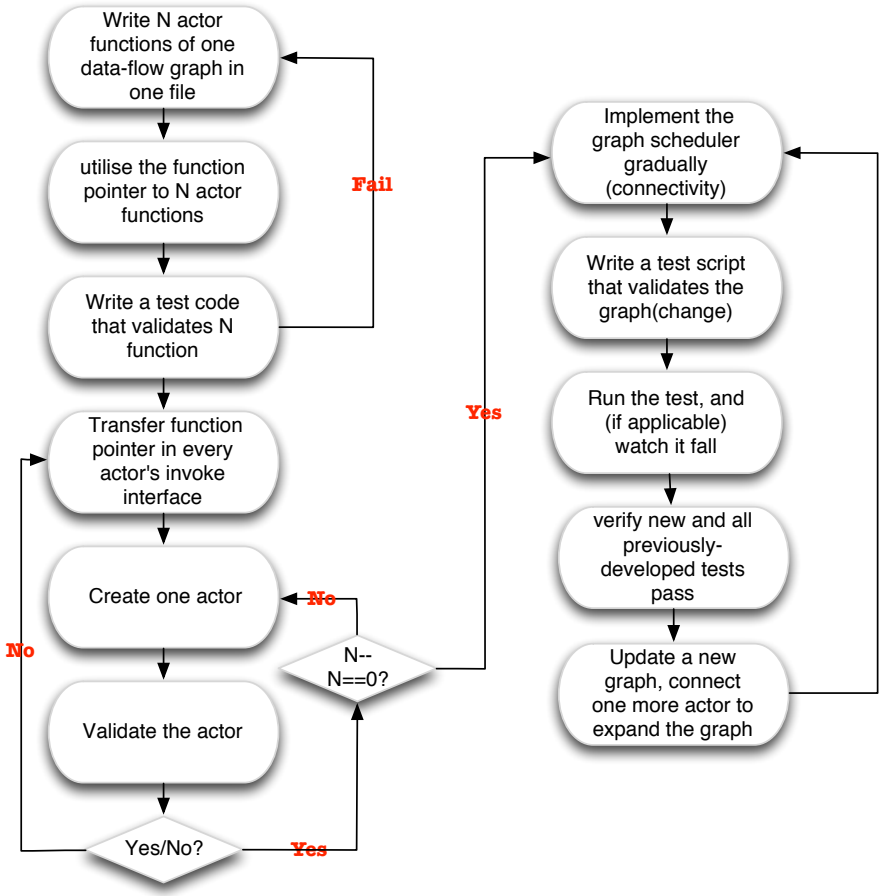


Figure 5.12: Flow diagram of LIDE-C code implementation.

Three designs has their own advantage. Design two and design three could be prepared for the future work and the following writing is majorly based on the design one.

5.3 Software Implementation Process

The process of implementation code is majorly made up of two steps. One is to create actors and the other is to connect the actors, set up the graph scheduler. Figure 5.12 describe the details of LIDE-C code implementation process.

5.3.1 Actor Code

From the chapter three's LIDE-C description, to creat one actor needs four key function and here it takes convolution actor implementation for example.

Figure 5.13 demonstrates the outline of construct function for this actor. The first three parameters of function, fifo_in, fifo_out, fifo_conv_wgt, have a relative

```
lide_c_image_convolution_context_type *lide_c_image_convolution_new(
            lide_c_fifo_pointer fifo_in, lide_c_fifo_pointer fifo_conv_wgt,
            lide_c_fifo_pointer fifo_out,int matrix_dimension, int coeff_dimension,
                lide_c_func_para_5 func)
{

    lide_c_image_convolution_context_type *context = NULL;

    context = lide_c_util_malloc(sizeof(lide_c_image_convolution_context_type));

    /* Initialize the CFDF mode of the actor */
    context->mode = LIDE_C_IMAGE_CONVOLUTION_LOAD_1;

    /* Setup the enable function for the actor */
    context->enable =(lide_c_actor_enable_function_type)lide_c_image_convolution_enable;

    /* Setup the invoke function for the actor */
    context->invoke =(lide_c_actor_invoke_function_type)lide_c_image_convolution_invoke;

    /* Setup actor parameters */
    context->func = func;
    context->matrix_dimension = matrix_dimension;
    context->coeff_dimension = coeff_dimension;

    /* Data specification */
    context->space_coeff = lide_c_util_malloc(sizeof(double) * coeff_dimension);
    context->space = lide_c_util_malloc(sizeof(double) * matrix_dimension);
    context->space_result = lide_c_util_malloc(sizeof(double) * matrix_dimension);

    /* fifo specification */
    context->fifo_in = fifo_in;
    context->fifo_out = fifo_out;
    context->fifo_conv_wgt = fifo_conv_wgt;

    return context;
}
```

Figure 5.13: Construct function for convolutional actor.

with the corresponding fifos, which is the dataflow edges, connecting to the actor port in one enclosing graph. The datum of fifo_in and fifo_conv_wgt's buffer are processed by the actor, and the result is produced and re-encapsulated into new buffer that fifo_output carries. Furthermore, the instantiation of one actor is also the initiation of the function pointers, like enable, invoke and lide_c_func_para_5.

Figure 5.14 reveals the outline of enable function for this actor. It is noted that not all of the actor ports would be needed during all of the CFDF modes. For example of this code, the mode of "LOAD_1" is only involved with fifo_in; "LOAD_2" is

```
boolean lide_c_image_convolution_enable(lide_c_image_convolution_context_type *context)
{
    boolean result = FALSE;

    switch (context->mode) {
        /* the prerequisite of loading image matrix */
        case LIDE_C_IMAGE_CONVOLUTION_LOAD_1:
            result = lide_c_fifo_population(context->fifo_in) < lide_c_fifo_capacity(context->fifo_in);
            result = result && (lide_c_fifo_population(context->fifo_in) >= context->matrix_dimension);
            break;

        /* the prerequisite of loading convolution weight */
        case LIDE_C_IMAGE_CONVOLUTION_LOAD_2:
            result = lide_c_fifo_population(context->fifo_conv_wgt) < lide_c_fifo_capacity(context->fifo_conv_wgt);
            result = result && (lide_c_fifo_population(context->fifo_conv_wgt) >= context->coeff_dimension);
            break;

        /* the prerequisite of doing convolution */
        case LIDE_C_IMAGE_CONVOLUTION_MODE_PROCESS:
            result = lide_c_fifo_population(context->fifo_out) <lide_c_fifo_capacity(context->fifo_out);
            break;

        default:
            result = FALSE;
            break;
    }
    return result;
}
```

Figure 5.14: Enable function for convolutional actor.

```
void lide_c_image_convolution_invoke(lide_c_image_convolution_context_type *context)
{
    switch(context -> mode)
    {
        case LIDE_C_IMAGE_CONVOLUTION_LOAD_1:
            /* Read enries of datum from fifo_in */
            lide_c_fifo_read_block(context->fifo_in, context->space, context->matrix_dimension);

            /* Switch Mode */
            context->mode = LIDE_C_IMAGE_CONVOLUTION_LOAD_2;
            break;

        case LIDE_C_IMAGE_CONVOLUTION_LOAD_2:
            /* Read enries of datum from fifo_conv_wgt */
            lide_c_fifo_read_block(context->fifo_conv_wgt, context->space_coeff, context->coeff_dimension);

            /* Switch Mode */
            context->mode = LIDE_C_IMAGE_CONVOLUTION_MODE_PROCESS;
            break;

        case LIDE_C_IMAGE_CONVOLUTION_MODE_PROCESS:
            /* do the convolution function */
            context->space_result = context->func(context->space, context->space_coeff,
                        (int)sqrt(context->matrix_dimension), (int)sqrt(context->coeff_dimension), SAME);

            /* Write the datum to the fifo */
            lide_c_fifo_write_block(context->fifo_out, context->space_result, context->matrix_dimension);

             /* Switch Mode */
            context->mode = LIDE_C_IMAGE_CONVOLUTION_LOAD_1;
            break;

        default:
             /* Switch Mode */
            context->mode = LIDE_C_IMAGE_CONVOLUTION_MODE_ERROR;
            break;
    }
    return;
}
```

Figure 5.15: Invoke function for convolutional actor.

just for fifo_out; "PROCESS" mode is exclusively related with fifo_out.

Figure 5.15 illustrates the outline of invoke function with an unconditional actor firing, because it is scheduler and enable function's responsibility to ensure there is enough resources to invoke the function. The invoke function calling without adequate datum and space would lead to unpredictable results. In the unit test process, this issue could be addressed.

Figure 5.16 elucidates the outline of terminate function. The target is to discharge the memories that has allocated during the construction and execution process.

5.3.2 Graph Scheduler

The steps of implementation on the graph scheduler are as followings:

(1) Create new actors and fifos.

(2) Allocate the buffers and space on these actors and fifos.

(3) Initialize these actors.

(4) Connect all the actors.

```
void lide_c_image_convolution_terminate(lide_c_image_convolution_context_type *context) {
    free(context->space);
    free(context->space_coeff);
    free(context->space_result);
    free(context);
}
```

Figure 5.16: Terminate function for convolutional actor.

```c
/* An enumeration of the actors in this application. */
#define ACTOR_READ           0
#define ACTOR_CONVOLUTION_1  1
#define ACTOR_CONVOLUTION_2  2
#define ACTOR_CONVOLUTION_3  3
#define ACTOR_ADDITION_1     4
#define ACTOR_ADDITION_2     5
#define ACTOR_MAXPOOLING_RELU 6

/* The total number of actors in the application. */
#define ACTOR_COUNT 7

int subgraph_conv1_SFM(...) {
    /* Preprocessing: open input file, check program usage... */
    ...
    /* Declaration actors and fifos */
    lide_c_actor_context_type *actors[ACTOR_COUNT];
    lide_c_fifo_pointer fifo1 = NULL, fifo2 = NULL, fifo3 = NULL, fifo4 = NULL,
    fifo5 = NULL,fifo6 = NULL, fifo7 = NULL, fifo8 = NULL, fifo9=NULL;

    /* actor descriptors (for diagnostic output) */
    char *descriptors[ACTOR_COUNT] = {"read", "convolution_1","convolution_2",
        "convolution_3","addition_1","addition_2", "maxpooling_relu","write"};

    /* Create the buffers and give the space. */
    token_size = sizeof(double);
    fifo1 = lide_c_fifo_new(BUFFER_CAPACITY, token_size);
    ...
    fifo9 = lide_c_fifo_new(BUFFER_CAPACITY, token_size);

    /* Create and connect the actors. */
    actors[ACTOR_READ] = (lide_c_actor_context_type*)(lide_c_file_read_channel_new(...);
    actors[ACTOR_CONVOLUTION_1] = (lide_c_actor_context_type*)(lide_c_image_convolution_new(...);
    actors[ACTOR_CONVOLUTION_2] = (lide_c_actor_context_type*)(lide_c_image_convolution_new(...);
    actors[ACTOR_CONVOLUTION_3] = (lide_c_actor_context_type*)(lide_c_image_convolution_new(...);
    actors[ACTOR_ADDITION_1] = (lide_c_actor_context_type*)(lide_c_image_addition_new(...);
    actors[ACTOR_ADDITION_2] = (lide_c_actor_context_type*)(lide_c_image_addition_new(...);
    actors[ACTOR_MAXPOOLING_RELU] = (lide_c_actor_context_type*)(lide_c_maxpooling_relu_new(...);

    /* Execute the graph. */
    lide_c_util_simple_scheduler(actors, ACTOR_COUNT, descriptors);

    /* Normal termination. */
    return 0;
}
```

Figure 5.17: Graph Scheduler Code.

(5) Run the schedule and execute the dataflow graph.

(6) Normal termination.

The sample code on dataflow of conv1_SFM(figure 5.6) is figure 5.17.

5.4 Functional Validation

Functional validation is a critical step for automatically validating the correctness of each implementation iteration before different transformations are applied. For this purpose, we apply the DSPCAD Integrative Command Line Environment (DICE), which provides language- and platform-agnostic features for testing of embedded signal processing software [36], [37].

Figure 5.18 illustrates the DICE-based organized, associated and systematic directory tree, which contain all hierarchy of the software and test modules for the DNN system design.

- All the sources (*.c,*.h,*.o) is contained in the directory "src".
- Every autotest-output directory depicts the current root's running situation.

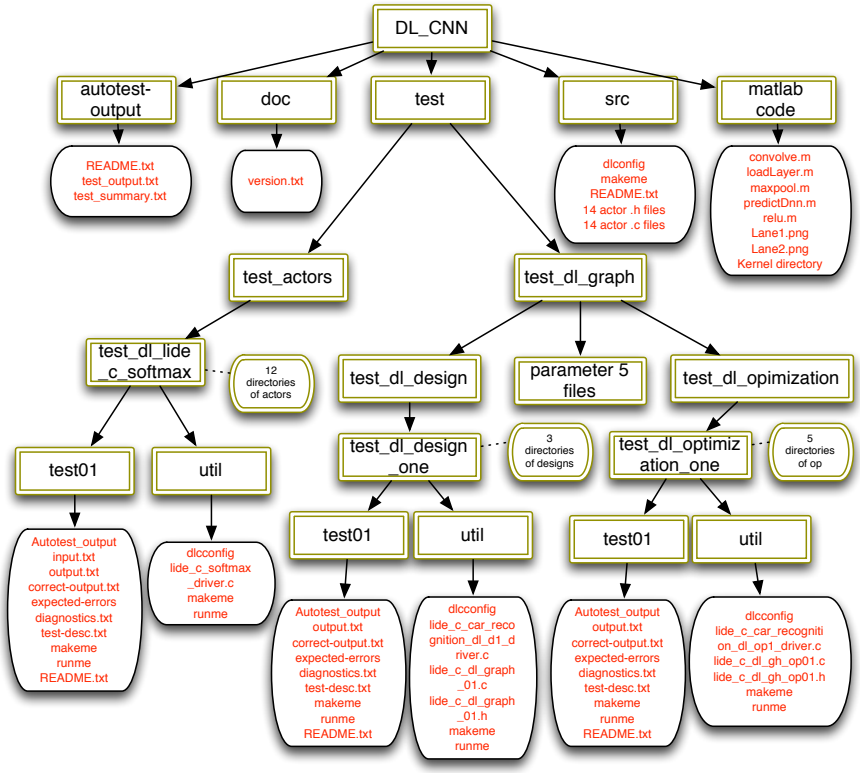


Figure 5.18: Directory tree of DL_CNN project.

- Test directory is divided into two section. On the thing is for the actors, and the other is for graphs.
- Every actor has its own test suite and individual directory to validate, verify itself.
- There are 12 actors (including optimized actor described later) in this dataflow graph and correspondingly produce 12 directories, and in every directory, there is at least one test sample to check the actor whether it is correct or not.
- After the validation on the actor, subgraph is needed to be establish with these validated actor before the final whole dataflow graph is completed based on these subgraphs, all which would be kept in the design directory. Three design patterns in this project produce three individual directories to record and store the files, which exert the compilation, validation's function.
- The process of optimization follows the design's step to update the designed dataflow graph. Similarly, the number of optimization has the number individual directories.
- There are 12 test suites of actors (unit test), 3 test suites of designs (system

test) and 5 test suites of optimizations (system test), 20 test suites all together. Only use "dxtest" command, all the 20 test suites would be tested, which saves an enormous amount of effort during the test validation process.

For further details on development and testing of signal processing systems using DICE, we refer the reader to [37].

6. DATAFLOW GRAPH TRANSFORMATIONS

After the complete of deep neural network design and implementation, we transform the whole DNN for the purpose of the better performance and efficiency. Figure 6.1 demonstrates our six transformations.

6.1 Transformation One : Broadcast Optimization

The transformation own "fork" function by means of creating the new actor (broadcast_token actor) after every conv1_SFM actor and the beginning of the read image datum. This behavior is to copy the buffer and pioneer another several fifos (threads) rather than to do repeat redundant operations from scratch, especially forking 32 fifos occurred between the first convolutional layer and the second convolutional layer. Figure 6.2(a) is the illustration of this "fork" optimization. The green diamond buffer is changed to red rectangle buffer after subgraph_conv1_SFM actor. In case the graph needs 32 pieces of the red rectangle buffer, the primitive graph makes it through repeat 32 times of the same operation. Therefore, it is fairly clever to create the broadcast_token actor, which exert the fork function, to avoid the thirty-two repeats of the same data-flow graph operation from the start point.

6.2 Transformation Two : Global Memory

With the consideration of relatively slow memory transfer (Read/Write I/O) in the computer system, the number of data transfer (load/store) between fifos should be as minimized as much as possible. Therefore, loading the convolution weight and other layer's parameters into global memory is another solution. The transformation has impact on every actor that is in possession of parameters/coefficients from imported datum. Figure 6.2(b) shows there are two fifo_in in the original way, which give rises to much more time for loading the datum compared to the after transformation two graph which has only one fifo_in to needed to load datum, plus putting convolution weight into global memory (imprint pointer).

6.3 Transformation Three : Multi-Addition Actor

This transformation is pointed to lots of lide_c_mtx_add actors. The idea is to use one multiple addition to replace these lots of additions. To take subgraph_conv2_SFM

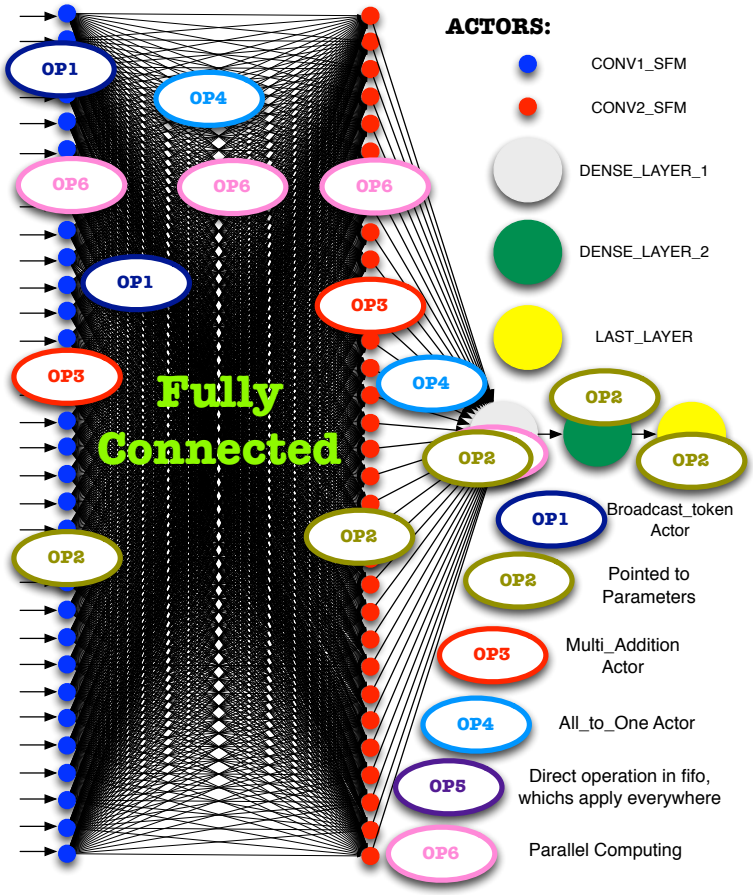
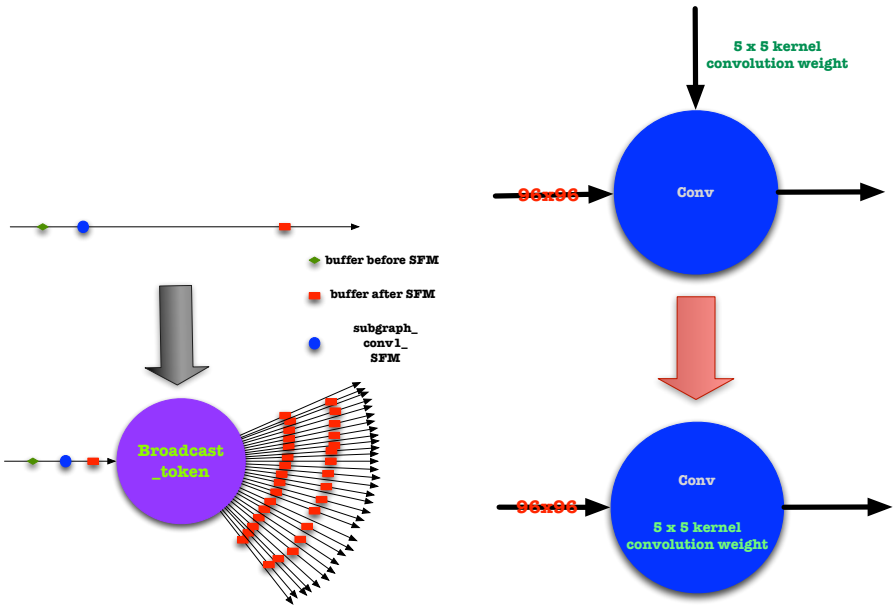


Figure 6.1: Transformation map.

as an example, there are 31 addition actors in that dataflow graph, which need much superfluous operations to load the fifo, do operation and then write the datum to fifo. It wastes some time on the 31 additions. The solution for that is to create a multiple_addition actor of dealing with a series of input additions once. Figure 6.3(a) is the illustration of this situation that addition actors (green) are replaced by one multiple_addition actor.

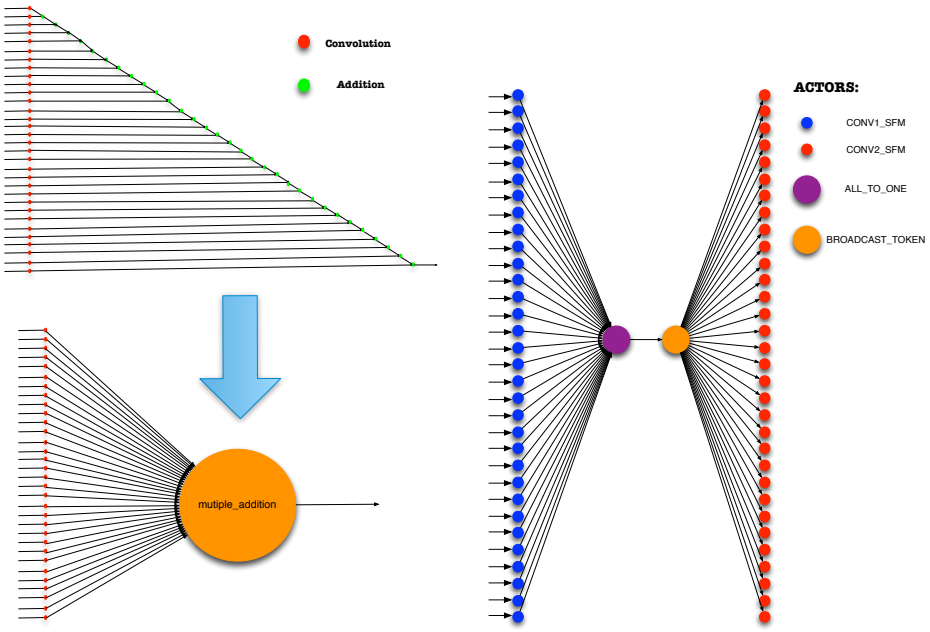
6.4 Transformation Four : Simplification of First Two Layers

From the view of design one, the transfer between convolutional layer one to layer two is a little bit heavy-worked. Too many links occurred there infer too much time to use for loading, operating, storing and transferring. Therefore, transformation four is to create a new actor (all_to_one actor), whose function is to assembly all input fifos into one fifo (decrease the dimension of datum). To add two actors (all_to_one actor and broadcast_token actor) between two convolutional layers could highly decrease the complexity of the dataflow and also the number of operations and actors. Figure 6.3(b) display the detailed dataflow graph of transformation four.



(a) Broadcast optimization. (b) Global memory optimization.

Figure 6.2: Transformation one and two.



(a) Multi-addition actor. (b) Simplification of first two layers.

Figure 6.3: Transformation three and four.

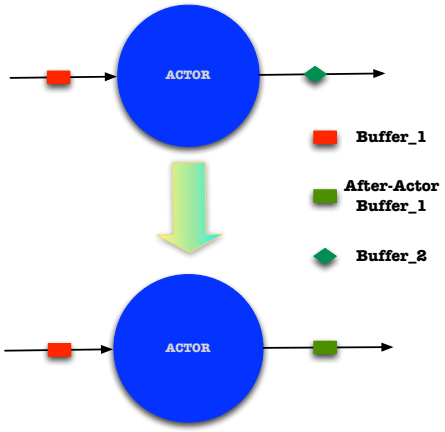


Figure 6.4: Transformation five : In-Place operations.

6.5 Transformation Five : In-Place Operations

Transformation five is to perform "in-place" operations on input data rather than having load mode. That means, instead of loading an image from an input FIFO into some internal storage for the actor, utilize the data directly from the input FIFO during the actor's computation, which will get rid of the associated LOAD mode, and the actor could run a lot faster. The transformation works on every actor in this dataflow graph. Figure 6.4 shows the details of transformation five. Shape represents container/fifo and color signifies the datum. The original one is to change the shape and color, which means to change the datum as well as fifo, need time to do two steps. Contrarily, after-transformed one is exclusively change the color, which exclusively changes the datum.

6.6 Transformation Six : Clustering into Threads

In this transformation, we cluster (or group together) subgraphs within the overall DNN dataflow graph to be executed as concurrent threads in the target multicore implementation [38]. This enables parallel execution of DNN subsystems when multiple cores are employed. Execution within the subgraph for each thread is managed by a LIDE-C-based dataflow graph scheduler that is dedicated to the thread, and the different schedulers for the different threads therefore execute concurrently for the overall DNN system. We employ pthreads as the interface for implementing the thread-based concurrent execution of the dataflow subgraph schedules[39],[40]. In our experimentation with alternative clusterings, we find that parallelization of 32 feature maps computations in the convolutional layers is especially effective in improving performance on the target platforms.

7. EXPERIMENTAL RESULTS

In this chapter, we separately operate the deep learning application in single core processor, test its performance in different environment, tools and conditions. Next, as a result of recent research [40] indicating the eventual optimizing effect would be still limited without concentration on the time-consuming operations, we run the after-parallelized DNN topology in the multiple core processors, respectively in the three different kinds of platform configurations that are targeted to three specific simulation situations: home application, remote application and embeded application.

7.1 Single Core Processor

Test items we focus on here are, the loading time - the time to load the DNN parameters obtained from the after-trained neural network, prediction time - the time required for classification of a single input image as soon as the training parameters has been loaded, and processing time - the addition of loading time and prediction time, which are measured from executing four versions of our DNN system concerning vehicle classification: (1) simulation implementation in matlab implementation; (2) unoptimized LIDE-C implementation; and (3) LIDE-C implementation with C compiler optimization - O3; (4) LIDE-C implementation with C compiler optimization - O3 and the dataflow optimization techniques discussed in chapter six. We refine the Fig. 7.1 from Table A.1, which are derived using a single-core, Intel Core i5-4248U processor with 8 GB memory and tested three times. The results indicate that applying model-based optimization techniques in LIDE-C significantly improves the overall single-core performance compared to the matlab simulation version, the unoptimized (initial) LIDE-C version and LIDE-C implementation with only C compiler optimization, while also exposing high-level dataflow structure that can be exploited to map the application onto multicore configurations in next step.

	Loading Time(sec)	Prediction Time(sec)	Processing Time(sec)
Matlab	76.1	3.6	79.7
LIDE-C (no optimization)	1.7	16.0	17.7
LIDE-C (compiler optimization)	1.6	13.0	14.6
LIDE-C (compiler and dataflow optimization)	1.5	0.8	2.3

Figure 7.1: Speedup results on single core.

7.2 Multi-Core Processors

Here, with the aid of the clustering transformation described in transformation six, we map the after-transformed LIDE-C implementation to parallelize the design for efficient execution on various alternative multicore configurations and run in the following plaforms.

Figure 7.2 is speedup results on the multiple cores, summed up from Tables A.2, A.3, A.4.

Table A.2 separately shows the loading time, prediction time, processing time in one core and two cores measured after mapping the optimized LIDE-C-based DNN implementation onto intel core i5 4248U. This situation could be applied in the home application with only one computer.

Table A.3 elucidate loading time, prediction time and processing time recorded three times, which are consumed in after-parallelized DNN and respectively run on one core, two cores, three cores, four cores, eight cores and twelve cores. This is remotely control case - a device that has an internet connection to one server, under the utilization of high performance on supercomputer clusters, remotely and automatically for pattern recognition.

Table A.4 reveals those loading time, prediction time and processing time in ARM A15 quad core (Odroid XU3 Board), and execute separately on one core, two cores, three cores, four cores. It could be worked as a embeded system - integrate the deep learning module into one device and off-line pattern recognition.

To sum up, these results quantitatively demonstrate the utility of after-transformed LIDE-C implementations in exploring complex design spaces for DNN systems involving alternative multicore platforms. From the results' view, the loading time could be invariant to how many cores the application runs on while the prediction

Platforms	Number of Cores	Loading Time(sec)	Prediction Time(Sec)	Processing Time(sec)
Intel core i5 4248U	1	1.52	0.80	2.32
	2	1.53	0.44	1.97
Two Six-core AMD Opteron 2435 Processor	1	0.59	0.41	1.00
	2	0.59	0.33	0.92
	3	0.59	0.27	0.86
	4	0.59	0.24	0.83
	8	0.59	0.20	0.79
	12	0.59	0.17	0.76
ARM Cortex-A15 quad core	1	3.06	2.09	5.15
	2	3.09	1.27	4.36
	3	3.07	1.03	4.10
	4	3.08	0.90	3.98

Figure 7.2: Speedup results on multiple cores.

time is consumed less and less with the utilization of an increasing number of cores. With the consideration of loading data incurred only once, at system setup time, the decrease of prediction time is a expected result - achieving further performance improvement by exploiting parallelism in these platforms.

8. CONCLUSIONS AND FUTURE WORK

This thesis is pertaining to the automatic classification on image-based vehicle recognition by deep neural network. We build on the DNN network structure derived in recent work on DNN-based vehicle classification, and we go beyond this previous work [1] by investigating aspects related to the efficient embedded implementation of this structure. Figure 8.1 shows a unified methodology for selecting, designing, establishing, modeling, mapping, transforming and validating deep learning architecture and implementations on resource-constrained platforms (FPGA implementation, the final two steps of Fig. 8.1, are the future work as is described later). With a view to iterative development of DNN implementation and optimization, we incorporate the lightweight dataflow environment (LIDE), which is a dataflow-based programming environment that allows signal processing system designers to apply and experiment with dataflow modeling approaches relatively quickly and flexibly in the context of existing design processes.

In particular, we employ LIDE-C, which is a part of the LIDE environment that is designed for use with C as the language for implementing dataflow-based software components (actors). LIDE-C provides application programming interfaces (APIs) that can be used when developing software modules using C such that the modules can be integrated together systematically as actors in an enclosing dataflow graph. This allows complete signal processing systems, such as our targeted DNN-based vehicle classification system, to be constructed as dataflow-based signal flow graph implementations where the actors are realized in C. Our use of LIDE-C in this thesis, as compared to other variants of LIDE, is motivated by the important role of C in embedded software implementation. Figure 8.2 is to summarize the process of LIDE-C design, implementation and optimization (LIDE-C Methodology). It is not only targeted to this thesis's application of LIDE-C programming, but also to any general application that is based on dataflow graph programming.

In this thesis, we have concretely demonstrated this process using a design and implementation case study of a deep neural network (DNN) for vehicle classification. Using the lightweight dataflow environment-C (LIDE-C), we have applied model-based design methods and using the resulting dataflow representations, we have applied a selected subset of design optimizations for the purpose of the performance improvement, and to derive efficient implementations of the targeted vehicle

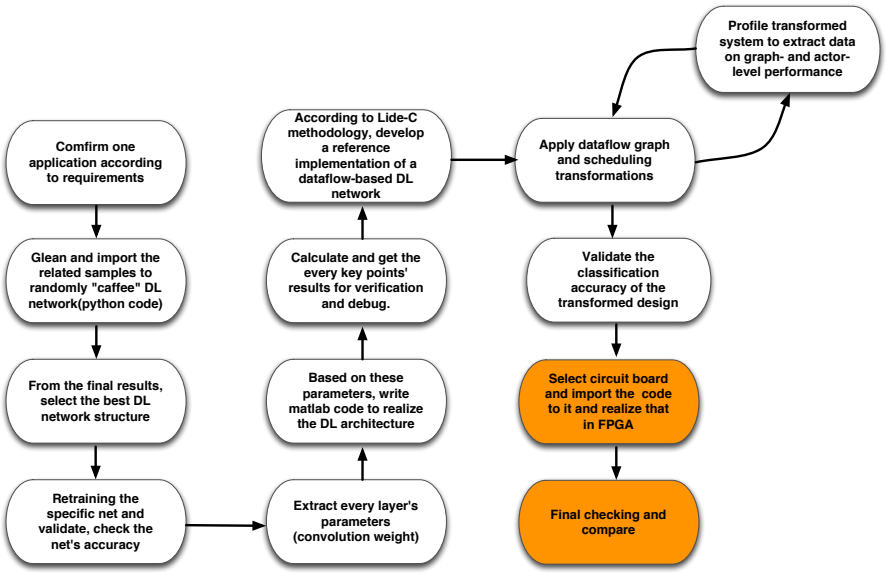


Figure 8.1: The methodology for the thesis.

classification system on three different multicore platforms with limited numbers of available cores. These transformations exploit the orthogonalization of actor implementation, task scheduling, and buffer management in LIDE-C, which allow for rapid prototyping of alternative implementation strategies for the given dataflow graph. While these transformations are not new design optimizations in and of themselves, their integration into resource-constrained multicore DNN implementations, and their application based on lightweight dataflow design principles are not only novel aspects of this thesis, but also getting the gratifying fruition.

However, the thesis is only the first version dataflow-based implementation on DNN application from scratch. Many sequential directions for future work include:

(1) Further optimization. It could be parallelized in the feature maps based on the design one. Supposed that digging further, you could also be parallelized in every convolution actors of every feature map, especially in the subgraph_conv2_SFM. Provided that digging thoroughly, it could be parallelized in every operation because every operation could be composed by product and addition, including convolution actor and matrix multiplication is also a typical application on computing in parallel, hence find internal parallelism of a certain algorithm, use lock-less asynchronous update to speed up the procedure. Therefore, parallel scheduler protocol may be another good and challenge topic. Furthermore, optimization of convolution actor - the most time used in the profiling DNN dataflow graph. Last but not the least, executing the loop tiling for the benefit of the next FPGA implementation is to solve the memory limitation in the hardware platform.

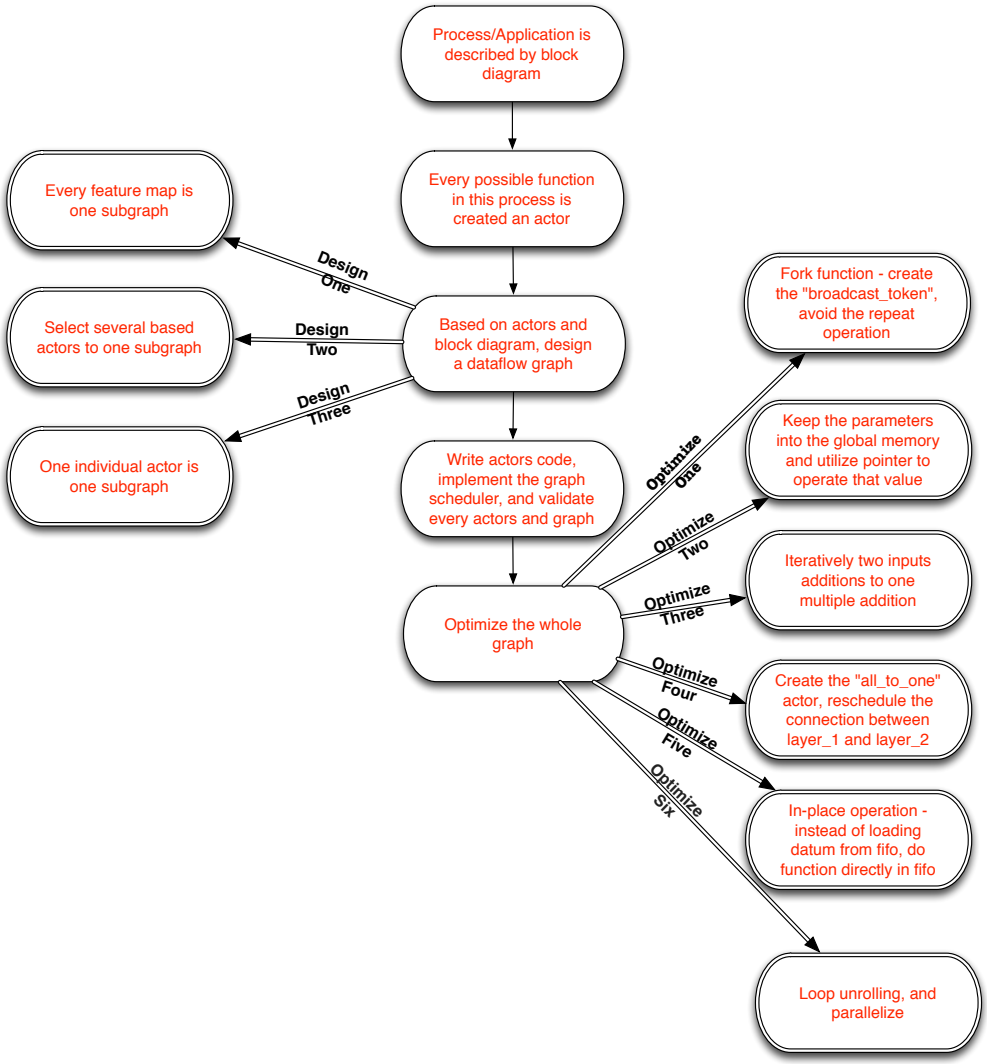


Figure 8.2: The process of LIDE-C programming.

(2) FPGA implementation. Choose one co-processor to be emphasized on how many cores, threads, tera-flops of performance and how large memory and bandwidth it provides, whether or not to support SIMD instruction or something. From the description, you could modify the code and make use of many techniques, like loop tiling, loop unrolling, loop pipelining, the size of selection, local memory promotion, loop transformations for data reuse according to CTC ratio and so on accelerator method, eventually match your board with least time.

(3) Dataflow-based deep neural network for training phase. By utilization of dynamic actor modes and parameterized scheduling of topology pattern, after forward-propagation and back- propagation training, the best combination of deep neural network would be searched and calculated - the exact number of

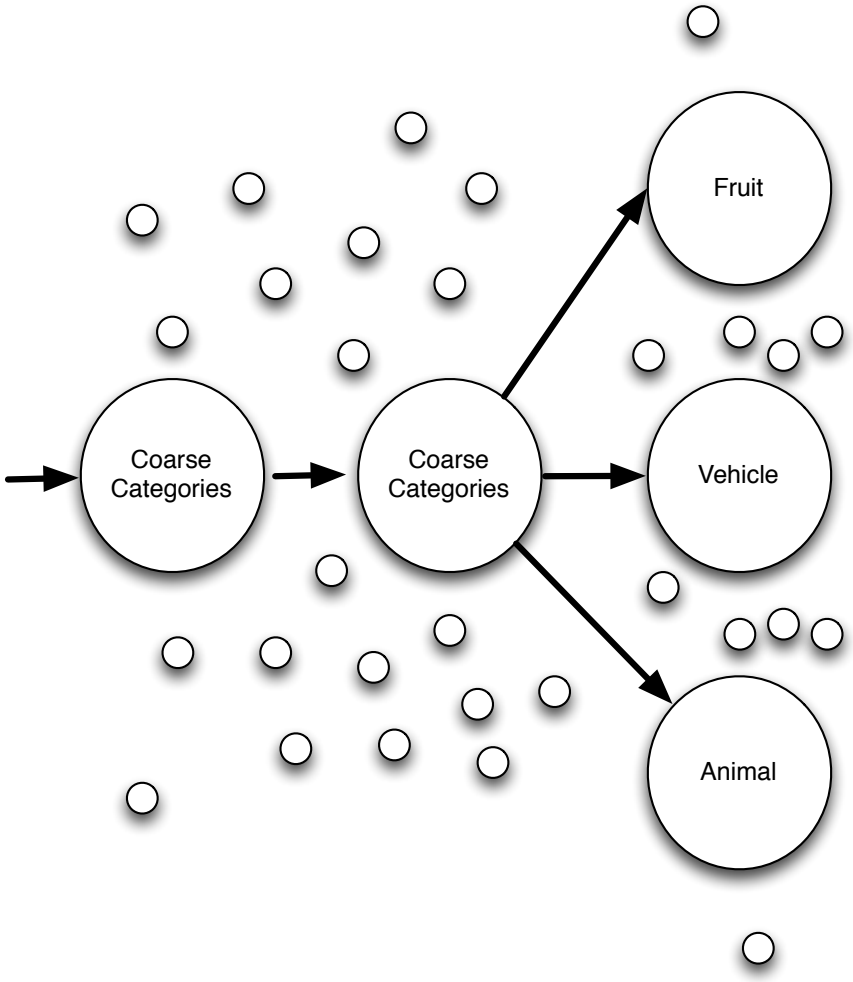


Figure 8.3: Future deep neural network.

layers and the number of neurons in each layer specific to one application.

(4) Establish the gigantic and complex DNN, not limited to the resources. Based on the methodology, accumulate different kinds of DNN according to your application and disciplinarily pipeline it to the huge the network. Figure 8.3 shows the rough and vague ideal DNNs, it goes through several coarse categories actor to the final refined DNNs; layer-to-layer actors are independent;buffers flow is dependable on the pevious actor results; Any actor consumed more time needs to be decomposed further. Therefore, implementation on different DNNs is the first step. Next is to connect these DNNS and finally optimized, reorder these DNN to breed up the final smart, gigantic and complex DNN.

The thesis is highly challenge and demanding, since it is vulnerable to be the worst consequence with the regard for the overall difficulty (understanding deep learning theory, pre-processing the large number of datum, designing dataflow graphs, implementing actor and graph codes, optimizing the dataflow graph and then implementation, finally testing in several platforms configuration, loop again and again for the best final result), especially no reference codes for such a large dataflow graph implementation. But completely controlling and torturing by myself, supporting from others and making progress day by day during the exhausting process let me gradually see the dawn of victory.

What went well this time is all the codes within the thesis are primitive, novel and complete, originating from my mind. Every programming language, testing platforms and compiling environment possess their own individual test directories, including descriptions, makefile, source codes, testing scripts. The beauty of that is researchers could continue the future job conveniently, and burn their brains to modify their codes for the purpose of the overall performance improvement, which could be occurred at any step (maybe the selection of DNN, the actor algorithm optimization, some specific platform application). However, there is still an imperfection that is I should further discuss with others or think deeper before the implementation of the whole dataflow graph, because some optimization could be solved more simply before implementation than after implementation with consideration of the large codes and graph, like float type instead of double type to save memory bandwidth for the prospect of hardware memory constrains. More discussions and many-sided thinking could make codes more robust, reliable, thorough and systematic at the start of the implementation.

BIBLIOGRAPHY

[1] H. Huttunen, F.S. Yancheshmeh, and K. Chen, "Car Type Classification with Deep Learning", ArXiv e-prints, February 2016, submitted to IEEE Intelligent Vehicles Symposium 2016.

[2] W. Zhao, R. Chellappa, P.J. Phillips, A. Rosenfeld, "Face Recognition: A Literature Survey", ACM Computing Surveys, Vol. 35, No. 4, 2003, pp. 399-458.

[3] Y. LeCun, C. Cortes, J.C. Burges, "THE MNIST DATABASE of handwritten digits", online: http://yann.lecun.com/exdb/mnist/.

[4] H. Li, H.F. Li, Y.T Wei, Y.Y Tang, Q. Wang, "Sparse-based neural response for image classification", Neurocomputing, vol. 144, p.198-207, November, 2014.

[5] A. Krizhevsky, "The CIFAR-10 Dataset", online: http://www.cs.toronto.edu/ kriz/cifar.html.

[6] Y.Taigman, M. Yang, M.A Ranzato, L. Wolf, "DeepFace: Closing the Gap to Human-Level Performance in Face Verification", In Computer Vision and Pattern Recognition (CVPR), June 24, 2014.

[7] J. Deng, W. Dong, R. Socher, L.J. Li, K. Li, and F.F. Li, "Imagenet: A large-scale hierarchical image database", In Computer Vision and Pattern Recognition (CVPR), 2009 IEEE Conference on, pages 248–255, 2009.

[8] D.E. Rumelhart, G.E. Hinton, and R.J. Williams, "Learning internal representations by error propagation", In David E. Rumelhart, James L. McClelland, and CORPORATE PDP Research Group, editors, Parallel Distributed Processing: Explorations in the Microstructure of Cognition, Vol. 1, pages 318–362. MIT Press, Cambridge, MA, USA, 1986.

[9] C.M. Bishop, "Pattern Recognition and Machine Learning", Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.

[10] D.E. Rumelhart, G.E. Hinton, and R.J. Williams, "Learning repre- sentations by back-propagating errors", Nature 323, pages 533–536, 1986.

[11] S. Haykin, "Neural Networks: A Comprehensive Foundation", Prentice Hall PTR, 2nd edition, 1998.

[12] Y. Bengio and A. Courville, "Deep learning of representations", In Monica Bianchini, Marco Maggini, and Lakhmi C. Jain, editors, Handbook on Neural Information Processing, pages 1–28. Springer, 2013.

[13] J. Schmidhuber, "Multi-column deep neural networks for image classification", In Computer Vision and Pattern Recognition (CVPR), 2012 IEEE Conference on, pages 3642–3649, 2012.

[14] M. Ranzato, "Supervised deep learning - tutorial on deep learning for vision", In Computer Vision and Pattern Recognition (CVPR), 2014 IEEE Conference on, 2014.

[15] A. Ng, A. Maas, A. Hannun, B. Huval, T. Wang, and S. Tandon, "Unsupervised feature learning and deep learning", Technical report, Stanford University, 2013.

[16] Help Conquer Cancer research team, "New imaging tools accelerate cancer research", online: http://www.worldcommunitygrid.org/about_us/viewNewsArticle.do?articleId=402.

[17] Petetin, C. Laroche, and A. Mayoue, “Deep neural networks for audio scene recognition”, in Proceedings of the European Signal Processing Conference, 2015, pp. 125–129.

[18] S.Ji, W.Xu, M.Yang, and K.Yu, "3d convolutional neural networks for human action recognition", IEEE Trans. Pattern Anal. Mach. Intell., 35(1):221-231, Jan. 2013.

[19] O. Gencoglu, T. Virtanen, and H. Huttunen, “Recognition of acoustic events using deep neural networks”, in Proceedings of the European Signal Processing Conference, 2014, pp.506–510.

[20] Krizhevsky, I. Sutskever, and G. E. Hinton, “ImageNet classification with deep convolutional neural networks”, in Proceedings of the Conference on Neural Information Processing Systems, 2012, pp. 1097–1105.

[21] C. Zhang, P. Li, G.Y. Sun, Y.J. Guan, B.J. Xiao, J. Cong, "Optimizing FPGA-based Accelerator Design for Deep Convolutional Neural Networks",23rd International Symposium on Field-Programmable Gate Arrays (FPGA2015).

[22] Y.Q. Jia, R. Girshick, S. Guadarrama, and T. Darrell, "Caffe: Convolutional architecture for fast feature embedding", arXiv preprint arXiv:1408.5093, 2014.

[23] Jia et al., “Caffe: Convolutional architecture for fast feature embedding”, arXiv preprint arXiv:1408.5093, 2014.

[24] S.S. Bhattacharyya, E. Deprettere, R. Leupers, and J. Takala, Eds, Handbook of Signal Processing Systems, 2nd ed. Springer, 2013.

[25] W. Plishker, N. Sane, M. Kiemb, and S. S. Bhattacharyya, “Heterogeneous design in functional DIF”, in Proceedings of the International Workshop on Systems, Architectures, Modeling, and Simulation, Samos, Greece, July 2008, pp. 157–166.

[26] E.A. Lee and D.G. Messerschmitt, "Synchronous Dataflow", Procceedings of the IEEE, vol.75, pp. 1235-1245,1987.

[27] G. Bilsen, M. Engels, R. Lauwereins, and J.A. Peperstraete, "Cyclo-Static Dataflow", IEEE Transactions on Signal Processing, vol.44, pp.397-408, 1996.

[28] B. Bhattacharyya, S.S. Bhattacharyya, "Paramterized dataflow modeling for DSP systems", IEEE Transactions on Signal Processing 49(10), 2408-2421, 2001.

[29] S.S. Bhattacharyya, E.F. Deprettere, and B.D. Theelen, "Dynamic Dataflow Graphs", online: http://www.es.ele.tue.nl/sadf/publications/HSPS12.pdf.

[30] W. Plishker, N. Sane, M. Kiemb, K. Anand, S.S. Bhattacharyya, "Functional DIF for rapid prototyping", In Proceedings of the International Symposium on Rapid System Prototyping, pp. 17-23. Monterey, California (2008).

[31] C. Shen, W. Plishker, H. Wu, and S.S. Bhattacharyya, "A lightweight dataflow approach for design and implementation of SDR system", In Proceedings of the Wireless Innovation Conference and Product Exposition, pages 640-645, Washington DC, USA, November 2010.

[32] C. Shen, W. Plishker, and S.S. Bhattacharyya, "Dataflow-based design and implementation of image processing applications", In L.Guan, Y.He, and S.-Y.Press, second edition, 2012. Chapter 24.

[33] C.C. Shen, L.H. Wang, I. Cho, S. Kim, S. Won, W. Plishker, and S.S. Bhattacharyya, "The DSPCAD Lightweight Dataflow Environment: Introduction to LIDE Version 0.1", Institute for Advanced Computer Studies, University of Maryland at College Park, USA, 2011.

[34] C.C. Shen, W. Plishker, and S.S. Bhattacharyya, "Dataflow-based Design and Implementation of Image Processing Applications", Institute for Advanced Computer Studies, University of Maryland at College Park, USA, May 23, 2011.

[35] N. Sane, H. Kee, G. Seetharaman, and S. S. Bhattacharyya, “Topological patterns for scalable representation and analysis of dataflow graphs”, Journal of Signal Processing Systems, vol. 65, no. 2, pp. 229–244, 2011.

[36] S. Kedilaya, W. Plishker, A. Purkovic, B. Johnson, and S.S. Bhattacharyya, “Model-based precision analysis and optimization for digital signal processors”, in Proceeding of the European Signal Processing Conference, Barcelona, Spain, August 2011, pp. 506–510.

[37] S.S. Bhattacharyya, W. Plishker, C. Shen, N. Sane, and G. Zaki, “The DSP-CAD integrative command line environment: Introduction to DICE version 1.1”, Institute for Advanced Computer Studies, University of Maryland at College Park, Tech. Rep. UMIACS-TR-2011-10, 2011.

[38] S. Kin and J.L. Pino, “Multithreaded synchronous data flow simulation”, in Proceedings of the Design, Automation and Test in Europe Conference and Exhibition, 2003.

[39] B. Nichols, D. Buttlar, and J.P. Farrell, "Pthreads Programming: A POSIX Standard for Better Multiprocessing", O’Reilly & Associates, Inc., 1996.

[40] L. Jin, Z.K. Wang, R. Gu, C.F Yuan and Y.H Huang "Training Large Scale Deep Neural Networks on the Intel Xeon Phi Many-core Coprocessor", IEEE 28th International Parallel & Distributed Processing Symposium Workshops, Nanjing University, Nanjing, China, 2014.

A. TEST RESULTS

A.1 Single Core on Four Design Versions

Intel Core i5 4248U				
Matlab (2014b)	(us)	#1	#2	#3
	Loading Time	77380000	76730000	76170000
	Prediction Time	3600000	3560000	3640000
	Processing Time	80980000	80290000	79810000
LIDE - C (no optimization)	Loading Time	1733112	1552176	1727277
	Prediction Time	16048042	16101079	16008110
	Processing Time	17781154	17653255	17735387
LIDE - C (compiler optimization)	Loading Time	1648571	1632924	1726092
	Prediction Time	14039306	13307456	12949667
	Processing Time	15687877	14940380	14675759
LIDE - C (optimzation and dataflow optimization)	Loading Time	1570659	1524946	1527219
	Prediction Time	895575	759665	804396
	Processing Time	2466234	2284611	2331615

Table A.1: Single core cost time on four design versions.

A.2 Multi-Core: Intel Core i5 4248U

Platform One : Macintosh Pro (Intel core i5 4248U)			
One Core			
(us)	#1	#2	#3
Loading Time	1570659	1524946	1527219
Prediction Time	895575	759665	804396
Processing Time	2466234	2284611	2331615
Two Cores			
Loading Time	1534027	1530938	1532227
Prediction Time	445137	438366	444084
Processing Time	1979164	1969304	1976311

Table A.2: Times measured on intel core i5 4248U.

A.3 Multi-core: Two Six-Core AMD Opteron 2435 Processors

Platform Two : Two Six-core AMD Opteron 2435 Processors			
One Core			
(us)	#1	#2	#3
Loading Time	589142	591966	590578
Prediction Time	415293	416249	415774
Processing Time	1004435	1008215	1006352
Two Cores			
Loading Time	589984	590331	597711
Prediction Time	326274	325289	333646
Processing Time	916258	915620	931357
Three Cores			
Loading Time	589604	596327	591131
Prediction Time	275682	270636	274971
Processing Time	865286	866963	866102
Four Cores			
Loading Time	590294	591800	595211
Prediction Time	240375	243722	252370
Processing Time	830669	835522	847581
Eight Cores			
Loading Time	594109	606795	592622
Prediction Time	214285	203654	205488
Processing Time	808394	810449	798110
Twelve Cores			
Loading Time	591995	601055	594560
Prediction Time	176902	178988	172799
Processing Time	768897	780043	767359

Table A.3: Times measured on two six-core processors.

A.4 Multi-core: ARM Cortex-A15 quad core

Platform Three : ARM Cortex-A15 quad core (Odroid XU3 Board)			
One Core			
(us)	#1	#2	#3
Loading Time	3077248	3055248	3090440
Prediction Time	2089504	2094901	2084905
Processing Time	5166752	5150149	5175345
Two Cores			
Loading Time	3102826	3109804	3091580
Prediction Time	1266860	1287002	1269194
Processing Time	4369686	4396806	4360774
Three Cores			
Loading Time	3078590	3072545	3075595
Prediction Time	1046456	1041778	1032368
Processing Time	4125046	4114323	4107963
Four Cores			
Loading Time	3103390	3115483	3084755
Prediction Time	905257	900453	911449
Processing Time	4008647	4015936	3996204

Table A.4: Times measured on ARM system.